OVERCOMING MEMORY CAPACITY CONSTRAINTS FOR LARGE GRAPH APPLICATIONS ON GPUS

A Dissertation
Presented to
The Academic Faculty

By

Prasun Gera

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

May 2021

OVERCOMING MEMORY CAPACITY CONSTRAINTS FOR LARGE GRAPH APPLICATIONS ON GPUS

Approved by:

Dr. Hyesoon Kim, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Santosh Pande
School of Computer Science
Georgia Institute of Technology

Dr. Tushar Krishna
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Moinuddin Qureshi
School of Computer Science
Georgia Institute of Technology

Date Approved: April 30, 2021

Live by the foma[1] that make you brave and kind and healthy and happy.

The Books of Bokonon 1:5

[1]Harmless untruths

For Pratik, who never loses sight.

ACKNOWLEDGEMENTS

A long time ago, I came across a manuscript about a mythical creature, the qilin. The tales were fantastic, and yet I had it on good authority that they were true. I wrote to one of the authors, Dr. Hyesoon Kim, to learn more about the said creature. This initial exchange led to a fruitful research relationship with her serving as my PhD advisor. She is as brilliant as she is kind, and this work would not have been possible without her guidance.

I would like to extend my gratitude to the thesis committee members: Dr. Richard Vuduc, Dr. Moinuddin Qureshi, Dr. Santosh Pande, and Dr. Tushar Krishna, for their valuable insights and feedback. I have learnt a great deal from them, both inside and outside the classroom. I would also like to thank Dr. David Bader for his support at a crucial time during my PhD. Special thanks to the late Dr. Sudhakar Yalamanchili for his kind support for my early work at Georgia Tech. His deep knowledge and enthusiasm were always a source of inspiration, and he will be dearly missed. I am grateful to Prof. Sundar Balasubramaniam and Dr. R. R. Mishra for their guidance and encouragement during my undergraduate years at BITS. I am forever indebted to my high school physics teacher and mentor, Mr. Kalpak Kothari, who taught me how to think.

The Hparch lab at Georgia Tech has been a second home over the years, and I would like to thank the current and past members - Hyojong Kim, Ramyad Hadidi, Pranith Kumar, Joo Hwan Lee, Dilan Manatunga, Nagesh Lakshminarayan, Jaewoong Sim, Lifeng Nai, Jen-Cheng Huang, Euna Kim, Bahar Asgari, Jiashen Cao, Andrei Bersatti, Blaise Tine, Yonghae Kim, and Jaewon Lee, for their support. I was also fortunate to have incredibly helpful mentors and collaborators - Sunpyo Hong, CK Luk, Alex Fender, Oded Green, and Daniel Lowell, during various internships. My stay in Atlanta has been memorable due to great friends - Rahul Agrawal, Piyush Sao, Robert Chen, Prashant Nair, Ankit Shrivastava, and Harsh Shrivastava.

I am deeply grateful to my parents and my brother Pratik for their unconditional love

and support. Last but not least, Srinivas Eswar has been a dear friend over the years, and our shared journey through graduate school has always been on the scenic route due to him. When he is around, one can't help but feel, "God's in His Heaven, All's right with the world".

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

GPUs have been used successfully to accelerate a number of graph applications. While remarkable in their raw compute throughput, GPUs are constrained by the capacity of local fast memory. Graphs arising from social networks, world wide web, phone networks, biological networks, etc. require tens to hundreds of gigabytes of storage even in common sparse representations, whereas GPU memory is typically in the order of a few gigabytes. As a result, the majority of prior work on graph applications on GPUs has been restricted to graphs of modest sizes that fit in memory. When faced with capacity constraints, the traditional approach in such problems has been to scale the problem to multiple compute nodes. The underlying assumption is still that the graphs fit in the collective memory of the nodes.

In this dissertation, we ask the following question: How can we accelerate graph applications on GPUs when the graphs do not normally fit in memory? This question opens up two lines of inquiry. First, GPUs are connected to the host over an interconnect such as PCI-e or NVLINK and have access to much larger, albeit slower, host memory as a part of a feature known as unified virtual memory (UVM). This forms the basis of the first problem wherein we seek to optimise the performance of graph kernels in this setting. Second, graph compression techniques have been proposed in a variety of different contexts. This presents an additional way of dealing with capacity constraints in which the application needs to decompress data on the fly during computation. However, most popular compression approaches do not map well to the GPU architecture. This forms our second problem wherein we seek to create a suitable compressed graph representation and efficient parallel graph decompression methods.

We formulate the first problem as a graph ordering problem and show that this formalism has a natural overlap with other graph ordering problems in literature. We propose a graph reordering method, HALO, that improves the locality of graph traversals in the

UVM context. HALO covers both directed and undirected graphs whereas prior methods only account for the latter case. We see a speedup of 1.5x-1.9x on reordered graphs in the UVM setting. We create an additional ordering method, BFS-BP, where we show that prior methods from the domain of graph compression such as recursive graph bisection can be suitably adapted to this problem. Next, we treat graph compression as a problem in its own right and describe a representation for compressed graphs that is amenable to run-time decompression on GPUs. We propose the Elias-Fano Graph (EFG) representation for graph compression based on the Elias-Fano encoding for monotonic sequences. We show that we can compress a variety of large graphs by a factor of 1.5x over the popular compressed sparse row (CSR) format. We decompose the seemingly irregular problem of runtime decompression into smaller high-performance primitives such as parallel scans and searches, which leads to an efficient and load-balanced implementation. We show that the runtime traversal performance for in-memory compressed graphs is 3.8x better than out-of-core implementations for CSR graphs. Further, our implementation is also 2x faster than the current state of the art in GPU based compressed graph traversals while maintaining a competitive compression ratio. Finally, we explore the interplay between graph reordering, graph compression, and performance.

CHAPTER 1
INTRODUCTION

Graphics Processing Units (GPUs) have been used successfully for accelerating graph based applications. These applications are both interesting and challenging for GPUs in that there is often ample parallelism in the algorithm, but the data access pattern tends to be highly irregular. Graph applications leverage thread-level parallelism and the high bandwidth afforded by the GPU's internal memory to hide long latency operations. Real world graphs tend to have sizes in the order of tens to hundreds of gigabytes. However, GPUs have traditionally only had accesses to local memory that is in the range of a few gigabytes. The majority of the prior work on GPUs and graph processing deals only with graphs that fit in the GPU's memory. While there is some work on distributed multi-GPU graph processing, the working assumption is still that the graphs fit in the collective memory of the GPUs.

GPUs are connected to the host via an interconnect such as PCI-E or NVLINK. Historically, data transfers between the host and GPUs have been the programmer's responsibility. Recent GPUs support unified virtual memory (UVM) between CPUs and GPUs. UVM simplifies a lot of programming abstractions by presenting a unified memory space to the programmer, and it also supports oversubscription of GPU memory. GPU memory can be oversubscribed as long as there is sufficient host memory to back the allocated memory. The driver and the runtime system handle data movement between CPUs and GPUs transparently (i.e., without the programmer's involvement). UVM allows us to run GPU applications that may otherwise be infeasible due to the large size of datasets. The performance impact of UVM, specifically with regard to oversubscription, has not been studied extensively. We found that using unified memory naïvely for graph applications leads to a severe performance penalty. With the increasing popularity of accelerators in solving do-

main specific problems, we believe that the problem of managing data movement efficiently between devices will become an important one in the future. Prior works that improve the locality or other memory access characteristics in graph applications span a few different approaches. On the theoretical side, there are external memory algorithms [1] that broadly model such systems. Some of these ideas have also been incorporated in disk based implementations [2]. A common theme in graph computing is to organise the underlying graph data in a particular way to improve performance. For example, graph partitioning is an important consideration in the distributed setting where the goal is to improve load balance and reduce communication costs. The data structures used for representing graphs also impact performance, and different graph structures [3, 4] as well as compression schemes [5] have been proposed. Graph ordering is another area with performance implications for graph applications and sparse linear algebra. Our focus in the first half of the dissertation is to preprocess static graphs by reordering them to improve the locality in a semi-external memory model such as one with UVM. We use breadth first search (BFS) as a representative graph traversal kernel. BFS has been studied extensively and is also equivalent to sparse matrix vector multiplication (SpMV) [6], which lets us compare relevant works from sparse linear algebra. It is also used as a building block in other applications such as betweenness centrality and strongly connected components. It captures the key properties of irregularity and unpredictability of future memory accesses, which are our main areas of interest in the UVM context. We found that prior reordering solutions such as RCM [7, 8], which generally perform well for sparse symmetric matrices, do not perform well for large directed graphs. Other prior methods such as Gorder [9] are too expensive to be feasible and also do not outperform RCM significantly for BFS. We propose a lightweight and effective a graph reordering solution, harmonic locality ordering (HALO), for improving the locality of graphs in typical traversal-like patterns. Additionally, we demonstrate that there is a natural overlap between the locality ordering problem and graph compression, and prior techniques from graph compression such as recursive graph bisection can be suitably

adapted to this problem.

Graph compression is a complementary approach that can be used for large graph problems. Traditional solutions for large graphs rely on distributed or out-of-core processing. The distributed approach has higher implementation complexity and hardware costs, and out- of-core processing is bottlenecked by the latency and bandwidth of the interconnect. With graph compression, one can accommodate larger graphs in GPU memory. The application works with compressed data and uses constant local memory to decompress only the needed portions at runtime. We implement breadth first search (BFS), single source shortest paths (SSSP), and PageRank (PR) for compressed graphs. The graph traversal pattern captures many fundamental challenges that arise in graph analytics on GPUs and has broad utility. While graph compression has been studied extensively in general [5], prior work has mostly focused on CPUs. This is partly because GPUs are still relatively new, but more so because the decompression stage is often sequential, irregular, and branch-intensive. At a high level, compression schemes reduce the storage requirements of repeating or predictable patterns, and one needs to follow dependent chains during decompression to recover the values. These characteristics do not map well to the GPU architecture. Graph analytics kernels on GPUs need to be written carefully to achieve high performance. It is important that threads in this architecture do the same *type* and *amount* of work without diverging too much. Skews in the degree distribution make the mapping of work to threads in the single instruction multiple threads (SIMT) model difficult. Compression adds another layer of imbalance to this since blocks of compressed data of the same size do not represent the same number of edges in the graph. Despite these challenges, graph compression is attractive since a GPU's internal memory bandwidth is an order of magnitude higher than the interconnect's bandwidth. In typical scenarios, graph analytics kernels are memory bound and the compute resources are under-utilised, which presents an opportunity to trade off compute resources for memory capacity. Our goal is to decompress the graphs at runtime as a part of the analytics kernel without severely affecting the performance. That

is, we need the decompression to be efficient enough that it can be overlapped with memory level parallelism afforded by the architecture. In the second half of the dissertation, we propose a GPU friendly graph compression format and techniques for efficient runtime decompression on GPUs.

1.1 Contributions

This dissertation makes the following contributions:

- We formulate the locality ordering problem in semi-external memory as an optimisation problem and show that it is NP-hard.
- We propose a new graph reordering method, HALO (Harmonic Locality Ordering), that targets data locality and volume of data transfer for traversal of large graphs on GPUs.
- Real world graphs are often directed and disconnected. These attributes make locality optimisations difficult for graph traversals. HALO improves the locality of arbitrary BFS traversals in a semi-external memory model such as one with GPUs and UVM.
- HALO performs better than prior methods such as RCM and can be parallelised as well as approximated efficiently. Traversals on reordered graphs show speedups in the range of 1.52x-1.9x. We also identify some optimisations that reduce unnecessary coherence traffic and lead to an additional 1.85x speedup.
- We create an additional ordering method, BFS-BP, that uses recursive graph bisection to solve the optimisation problem formulated earlier.
- We present a compressed graph representation, Elias-Fano graph (EFG) format, based on the Elias-Fano encoding scheme for monotone sequences. Our contribution here

is the GPU implementation[1] for run-time decompression in graph analytics. We implement BFS, SSSP, and PageRank for compressed graphs.

- The key challenge in efficient run-time decompression on GPUs is maintaining a high degree of parallelism and partitioning the work evenly between threads. This is achieved by decomposing the problem into smaller high-performance primitives such as parallel scans and searches.

- The proposal satisfies several desirable criteria such as high decompression throughput, competitive compression ratio, a priori determination of compression ratio, independence from the graph's ordering, and low compression time.

- EFG achieves an average compression ratio of 1.5x over the compressed sparse row (CSR) format, and breadth first traversals show a speedup of 3.8x over the equivalent out-of-core CSR implementation.

- Our implementation also achieves a speed-up of 2x over the current state of the art in GPU based compressed graph traversals by Mo Sha et al. [10] (SIGMOD '19).

- We show that the EFG encoding is tolerant to pathological graph orderings. While preprocessing is not needed for EFG to achieve a good compression ratio, locality friendly orderings can improve the decompression performance further.

1.2 Organisation

The dissertation is organised as follows:

In Chapter 2, we introduce the unified memory architecture and its performance characteristics in the context of graph traversals. We review breadth first search and identify the main issues that contribute to poor performance.

[1] `https://github.com/pgera/efg`

In Chapter 3, we formalise the locality based graph ordering problem as the MinIntraBFS problem and prove that it is NP-hard. We also review other related graph ordering problems and prior work in the area.

In Chapter 4, we show detailed measurements based on hardware and software performance counters that shed light on the underlying inefficiencies in the UVM model.

In Chapter 5, we describe two new graph reordering methods, Harmonic Locality Ordering (HALO) and BFS-BP, that improve the performance of arbitrary traversals on general directed graphs in the UVM setting.

In Chapter 6, we evaluate the effectiveness of different ordering methods. We discuss the results in detail and show that performance improvements can be tied back to both profiling data and the objective function from the MinIntraBFS problem.

In Chapter 7, we motivate the graph compression problem and describe the Elias-Fano encoding scheme which forms the basis for our compressed graph representation.

In Chapter 8, we propose the Elias-Fano graph representation and describe a load-balanced GPU implementation for run-time decompression in graph analytics. We also review prior work in the area.

In Chapter 9, we evaluate the proposed graph representation in terms of compression ratio and decompression time. We also explore the relationship between graph reordering, compression, and performance.

Chapter 10 concludes the dissertation.

CHAPTER 2

TRAVERSALS IN UNIFIED MEMORY

2.1 Motivation

In recent years, several works have proposed methods for doing efficient graph processing on GPUs. Breadth first search (BFS) in particular is often studied as a representative kernel and is also included in benchmarks such as Parboil [11], Rodinia [12] and the Graph500 [13] benchmark. The core BFS kernel is also used as a building block in other graph analytics applications such as betweenness centrality [14] or strongly connected components [15]. Several prior works [16, 17, 18, 19, 20, 21] and GPU graph frameworks [22, 23, 24] cover different optimisations that deal with issues arising due to load imbalance, uncoalesced memory accesses, or redundant work. The majority of prior work on graph applications on GPUs assumes that the graph fits in the GPU's memory. For larger graphs, there are some works that distribute the graph over multiple GPUs [25, 26] or between the CPU and the GPU [27].

Recent GPUs support unified virtual memory (UVM) between multiple GPUs and CPUs. Support for this feature has made it into language specifications such as CUDA [28] and OpenCL [29], as well as vendor implementations and kernel drivers [30]. UVM presents a unified abstraction for memory management between several devices, and it supports oversubscription of device memory. The driver manages on-demand migration of pages transparently between devices. UVM is attractive for development since the changes required are generally not very invasive for existing applications, and it makes working with datasets larger than a single GPU's memory feasible. While applications such as deep learning regularly deal with datasets larger than the GPU's memory, they are explicitly designed to work in a pipelined fashion where small batches of data are sent to the GPU at

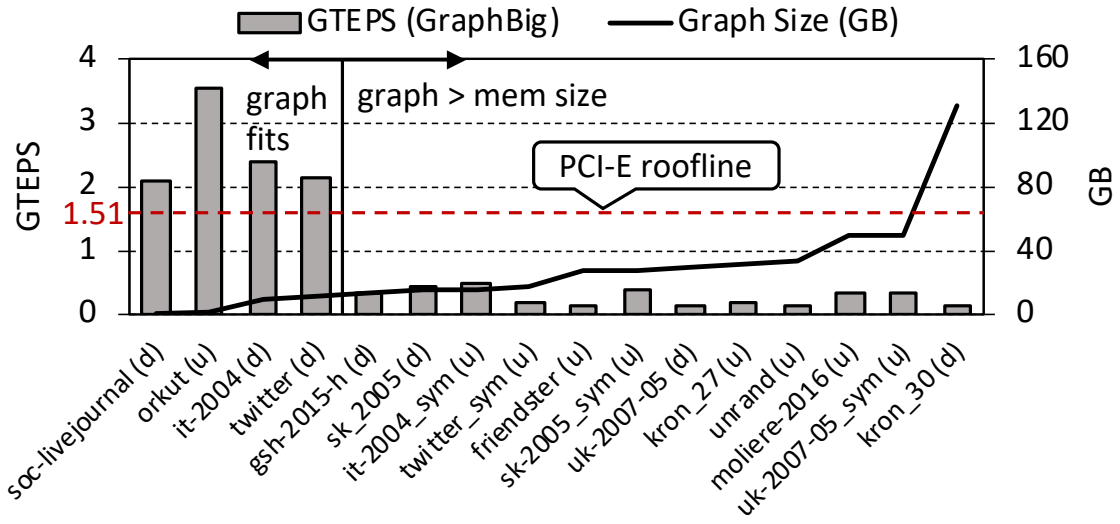


Figure 2.1: BFS performance measured in billions of Traversed Edges per Second (GTEPS) for a Titan Xp GPU

Table 2.1: GPU Bandwidth Characteristics

GPU	Memory Capacity	HtoD Link	DtoD Bandwidth	HtoD Bandwidth
Titan Xp	12 GB	PCI-e 3.0	417.4 GB/s	12.1 GB/s

a time which can be overlapped with computation. This approach does not map naturally to irregular graph traversal kernels since the data that would be needed in the future is not known beforehand. Further, an application like BFS has low computational intensity making it difficult to overlap computation with data transfers. Hence, most prior work on GPUs and graph computing deals only with sizes that fit in the GPU's memory. UVM makes it possible to use existing GPU graph frameworks for large graphs without extensive changes.

While UVM makes it easy to overprovision memory, the performance is quite poor if we take an application like BFS and change its `cudaMalloc` calls to `cudaMallocManaged` calls. In Fig. 2.1, we show the BFS performance results, measured in billions of traversed edges per second (GTEPS), across different graph sizes after minimally modifying GraphBig [24] to take advantage of UVM. All the data structures are allocated with UVM. The results are accumulated over fifty traversals from random sources for each graph. We see

that performance drops off sharply when the graph sizes exceed the GPU's memory capacity of 12 GB. The Titan XP card uses the PCI-E 3.0 interface with a peak bandwidth of 12.11 GB/s. We use 64-bit data types, which gives us a theoretical peak GTEPS rate of 1.51. This is a theoretical rate for just transferring data linearly whereas an application like BFS does highly irregular work as well. We see that the average GTEPS rate for BFS in the UVM region is 0.24, which is much lower than the theoretical peak, and we would like to improve its performance. It is also worth noting that this is a memory bound problem where current optimised parallel CPU implementations [31] will outperform UVM due to higher DRAM bandwidth. However, faster interconnects like NVLINK have much higher bandwidth, and newer versions of PCI-E as well as other interconnects like OpenCAPI are already making their way to products. Additionally, the traversal itself may be a part of a larger GPU accelerated pipeline with higher computational complexity. Newer architectures with non-volatile memory or storage attached directly to CPUs or GPUs will face similar challenges. We view this as a forward looking problem that motivates and prepares for advances in memory systems, interconnects as well as heterogeneous computing. To that end, we would like to answer the following questions:

- What are the primary factors that impact the performance of large graph traversals in the UVM model?
- How is the UVM memory hierarchy different from other hierarchies?
- How can we improve the performance of large graph traversals in the context of UVM?
- How can we compress graphs such that analytics on compressed graphs are still parallel and efficient on GPUs?

2.2 Background

2.2.1 Graph Storage

We use the compressed sparse row (CSR) data format for representing graphs. Since real world graphs tend to be sparse, the CSR format, which stores the non-zero elements instead of the entire adjacency matrix, is a suitable choice. The CSR format is also popular for static graphs as one can leverage high performance sparse linear algebra libraries. Dynamic graphs, which are the outside the scope for this work, would benefit from a different representation such as the one used by Hornet [32]. In the CSR data format, a graph $G = (V, E)$ with node set V and edge set E is stored as three arrays. Through the rest of the paper, we use the common convention of denoting $|V|$ as n and $|E|$ as m. The first array, `vlist`, is of length $(n + 1)$. It consists of row offsets that are used to index the second array. The second array, `elist`, is of length m and contains the column indices. For a given vertex id v, its neighbours can be accessed in the range `elist`[`vlist`[v] : `vlist`[$v + 1$]]. A third optional array of size m can be used for storing edge weights. An application such as BFS may use additional data structures for storing the output or intermediate information. In our implementation, BFS uses an array `vprop` for storing vertex property information such as the level number in the traversal. Throughout this work, we use 64 bit types for `vlist`, `elist` and `vprop` so that the work accounts for the most general massive graphs. In practice, it is possible to use either 32 bit types or a combination of 32 bit and 64 bit types for smaller graphs. Figure 2.2 shows a sample graph and its corresponding CSR representation.

2.2.2 BFS

Breadth First Search (BFS) is a fundamental graph traversal algorithm that is used as a building block in other graph algorithms. The algorithm starts graph traversal from a given source and visits other vertices in the graph in level order. The algorithm has a serial

complexity of $\mathcal{O}(n+m)$. BFS has been studied extensively, and there are optimised implementations for GPUs [16, 17, 18, 19, 20, 21]. Further, the problem can also be expressed as an iterative sparse matrix vector multiply problem (SpMV) [33]. A simple GPU kernel that traverses one level in a top-down BFS is presented in Alg. 1. This is called from the host until `stop` is false, at which point the BFS tree has been completely explored. Levels are initialised to ∞ for all vertices except the source, which is initialised to 0. We have omitted the predecessor computation for brevity. A central part of the BFS algorithm is the notion of a frontier. The vertices in the current frontier explore their neighbours and add them to the next frontier. This step at line 6 in the algorithm is responsible for bringing in data from the CPU to the GPU in the UVM setting. Note that the level array is serving as an implicit frontier here, which is also the vector when you see BFS as SpMV. It is possible to use different data structures for the frontier along with different strategies for load balancing the exploration of edges going out of a frontier. These considerations are orthogonal to the UVM model and can be applied independently. We are mainly concerned with memory accesses in the `elist` array at line 6 which remain unchanged as they are a function of the graph's topology and ordering.

Algorithm 1 BFS_Advance (G, level, curr, stop)

```
1:  for all v ∈ V in parallel do
2:      if level[v] == curr then
3:          start_idx = vlist [v]
4:          end_idx = vlist [v+1]
5:          for all idx ∈ [start_idx, end_idx) in parallel do
6:              nbr = elist[idx]
7:              if level[nbr] == ∞ then
8:                  level[nbr] = curr + 1
9:                  stop = false
10:             end if
11:         end for
12:     end if
13: end for
```

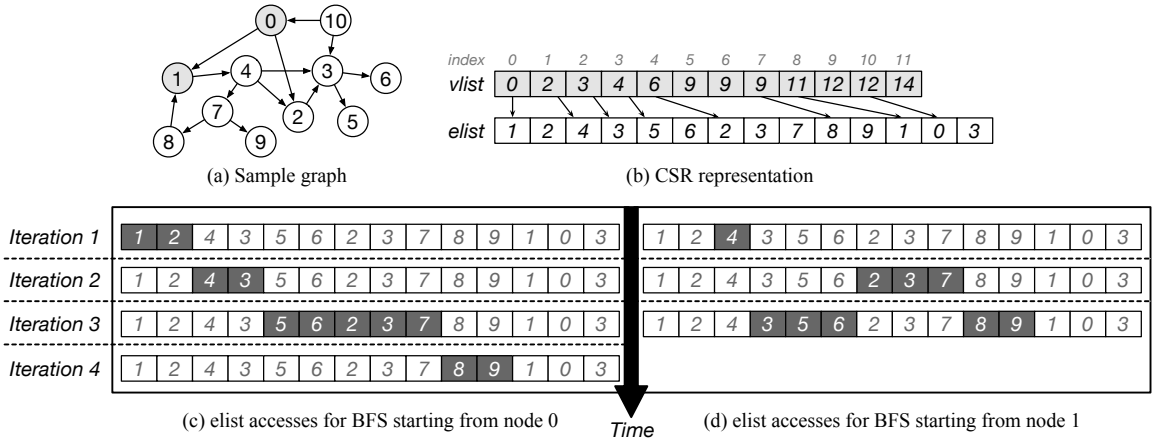
2.2.3 Model Assumptions

Our primary working model for UVM is a semi-external memory one. We cover graphs whose $\mathcal{O}(n)$ structures such as `vlist` and the level array can fit in the GPU memory whereas $\mathcal{O}(m)$ structures such as `elist` do not. While unified memory does not preclude much larger graphs where even these structures do not fit in GPU's memory, such problems will likely need additional work to achieve efficient solutions. For general external memory BFS, we refer the reader to theoretical work in the area [1]. With current GPU memory capacity trends, the semi-external model still scales to billion node graphs. All the graphs are treated as unweighted since we are primarily concerned with BFS.

2.3 Data Access Pattern

Since the graphs that we want to work with are larger than the GPU's memory, data is transferred between the host and device on demand at runtime. The bandwidth between the host and the GPU is much lower than the GPU's internal bandwidth. In Table 2.1, we can see that the external bandwidth is 34X lower than the internal bandwidth. Further, the effective bandwidth is inversely proportional to the chunk size of transfers. Since unified memory is a page fault handling mechanism, transfers happen at page granularity (4 KB) or multiples thereof. The runtime uses prefetching and batch processing of page faults to mitigate the high latency of transfers [34, 35, 36, 37, 38, 39, 40, 41]. However, this has side effects such as read/write amplification, thrashing, and evictions which are exacerbated in cases of irregular applications like graph traversal. We look at some simple examples in our sample graph from Fig. 2.2 to highlight the factors that impact performance.

Dependence on the Source Node: Consider a BFS from source node 0 in the sample graph in Fig. 2.2. We assume that elist is larger than GPU memory and paged in on demand from the host. We show the progress of the traversal in 2.2(c). The shaded blocks in elist are the ones that are accessed at each level in line 6 of Alg. 1. Let us also consider a BFS



(a) Sample graph (b) CSR representation

(c) elist accesses for BFS starting from node 0 (d) elist accesses for BFS starting from node 1

Figure 2.2: A sample graph, its CSR representation, and BFS access pattern from different sources

from a different source node 1, whose accesses are also denoted in the 2.2(d). The first BFS from node 0 is the ideal case for UVM as all the accesses in `elist` move contiguously in one direction with time. This leads to fewer page transfers, good page replacement decisions and better prefetch performance due to the predictability of accesses. The second traversal from node 1, however, access non-contiguous locations. This is a small example, but in a large graph, such accesses could span several pages, leading to multiple page transfers with a low ratio of useful data, thrashing and poor accuracy for the prefetcher. We make the following observation: The source node in a BFS affects the locality of accesses. We have no control over the source. Our goal is to optimise traversals from arbitrary source nodes.

Dependence on Graph Ordering: Consider a different ordering of the same sample graph as show in Fig. 2.3 and a BFS traversal from source node 0. The access pattern of words in elist is very different from the contiguous pattern in Fig. 2.2(c) although it is the same traversal. We make the following observation: A graph's ordering affects the locality of accesses in a BFS traversal. If a frontier's nodes are close to each other in their labels, as they are in Fig. 2.2(c), the exploration of their neighbourhoods would have good locality in `elist` in the next iteration.

Dependence on Directedness: Directed edges and the structure of the graph affect lo-

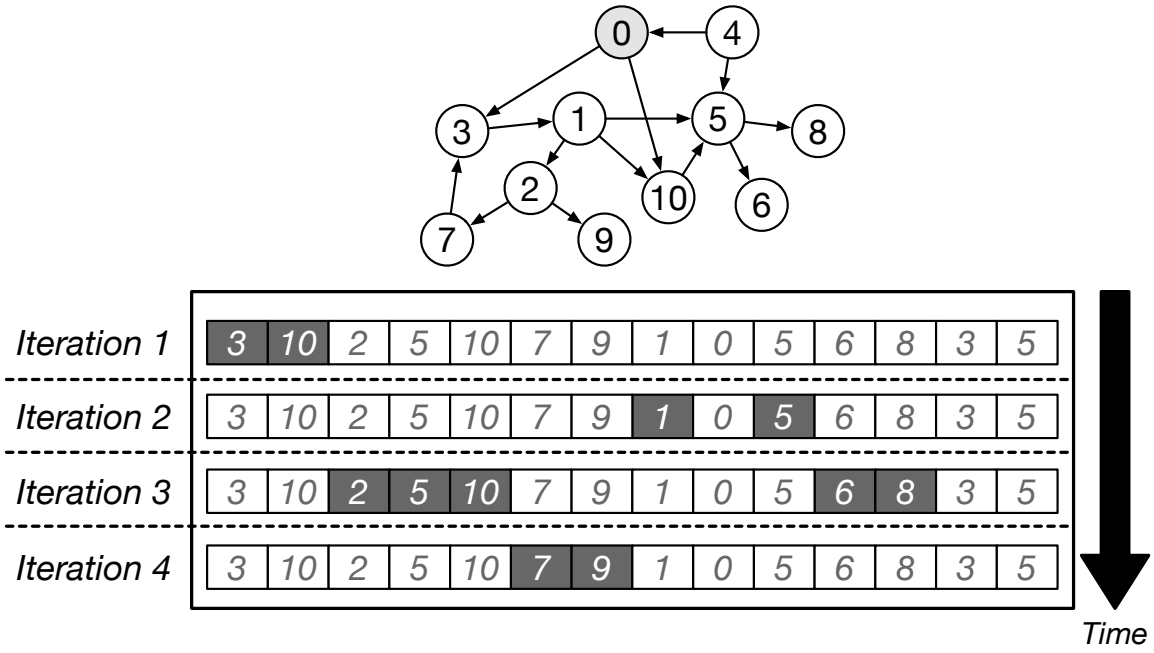


Figure 2.3: `elist` accesses for BFS from node 0 in an alternate graph order for the same graph as Fig. 2.2

cality of accesses in `elist`. For example, a traversal from node 5 in the original graph would not explore any additional nodes, but would traverse the entire graph in the undirected version since there is only one connected component. This is an important distinction because an ordering that optimises undirected graphs does not necessarily optimise directed ones. This is also why degree based metrics are inadequate for directed graphs. The degree of a node in an undirected graph can be a good measure of connectedness and reachability. However, in directed graphs, not only do we have separate in and out degrees, neither is a good indicator of broader reachability since the degree only looks at connectivity one hop away. For example, a node may have high in-degree and out-degree, but if we look beyond the immediate neighbours, this node may not be reachable from portions of the graph at all.

Table 2.2: Graphs used in this work

Graph	Category	Vertices	Edges
soc-livejournal (d)	Social Network	4.85 M	68.99 M
orkut (u)	Social Network	3.07 M	234.37 M
friendster (u)	Social Network	65.61 M	3.61 B
twitter (d)	Social Network	41.65 M	1.47 B
it-2004 (d)	web crawl	41.2 M	1.15 B
gsh-2015-h (d)	web host graph	68.66 M	1.80 B
sk-2005 (d)	web crawl	65.61 M	1.95 B
uk-2007 (d)	web crawl	105.22 M	3.74 B
moliere-2016 (u)	Bio Hypothesis	30.22 M	6.68 B
kron_(27,28,29,30)	kronecker	2^{scale}	$\|V\|$ x 16
unrand (u)	Erdos-Renyi	134.22 M	4.29 B

2.4 Graph Selection

We use several large real world and synthetic graphs spanning web crawls [42, 43, 44], social networks [45] and a biological hypothesis network [46]. These graphs generally have a small diameter, high clustering, and scale-free degree distributions. As a counterpoint to such graphs, we also include one uniformly random Erdos-Renyi graph [47], where our goal is to show that locality based optimisations are dependent on the graph structure and do not extend to graphs such as uniformly random graph. Our graphs include a mix of directed and indirected graphs. For the directed graphs, we also show results for symmetrised version of the same graph. These graphs are noted with the *sym* suffix. Directed graphs are noted with (d) whereas undirected ones are noted with (u).

CHAPTER 3

PROBLEM FORMULATION

Given that BFS has a low computational complexity, we are primarily memory bound, and the inefficiencies of the UVM model indicate that from an algorithmic standpoint, the main avenues for improvements in this area are reducing the volume of data transfer and improving the efficiency of transfers. There are broadly three ways to improve the performance:

1. Change the algorithm to do less work
2. Reorder the graph
3. Compress the graph

Direction optimising BFS [20] is a solution in the first category that reduces the number of edges visited. Such solutions can be applied independently. We cover the reordering problem here and cover graph compression in Chapter 7. Based on the observations in Sec. 2.3, there are two desirable properties that we seek in an ordering:

1. The labels of nodes *within* a frontier are close to each other
2. The labels of nodes *across* successive frontiers are close to each other

When combined, these two properties lead to a pattern similar to the one we saw in Fig. 2.2(c). For the first property, we formulate the problem as follows: For a general directed unweighted graph, we define the MinIntraBFS ordering as the ordering that minimises the total objective

$$C = \sum_{\text{sources}} \sum_{\text{levels}} \sum_{i=1}^{|\text{level}|-1} c(\pi(u_{i+1}) - \pi(u_i)) \tag{3.1}$$

where π is the ordering function, $u_1..u_{|level|}$ are the nodes in each level of the traversal such that $\pi(u_i) < \pi(u_{i+1})$, and c is a cost function. That is, we seek to lower the gaps between node labels *within* any frontier from any source node in the graph. This formulation ends up being quite close to other graph ordering problems such as the minimum linear (or $\log$) arrangement problem (MinLinA/MinLogA) [48, 49] and the minimum linear (or $\log$) Gap Arrangement (MinLinGapA/MinLogGapA) [49] problems. The MinLinA problem, which is the oldest of these problems, was originally suggested as a model for VLSI layout optimisation. The goal in MinLinA is to find a permutation for vertex labels that would reduce the total length of edges if the graph were laid out on a linear axis with vertices pegged at their respective labels. The objective function in MinLinA is $\sum_{(u,v)\in E} |\pi(u) - \pi(v)|$. The gap based variations on the other hand are motivated by graph compression. They aim to reduce the gaps between labels of neighbours so that neighbour lists can be compressed efficiently. The neighbour lists are represented as a list of successive gaps, and the gaps are further encoded using an efficient code. The original list can be recovered by computing a prefix sum over the gaps. Since smaller gaps are preferable for encoding, the overall objective is to minimise such gaps. While the linear versions of these problems assume an identity cost, the $\log$ versions are motivated more directly by the binary encoding of gaps. All these problems are known to be NP-hard. Recently, Dhulipala et al. [50] proposed a new model called the bipartite minimum logarithmic arrangement (BiMLogA) problem that generalises both MinLogA and MinLogGapA. If the cost function c is identity or $\log$, we can show that MinIntraBFS problem is also NP-hard.

3.1 MinIntraBFS

Theorem 1. *MinIntraBFS (log cost) is NP-hard*

Proof. We prove this by reducing an instance of the BiMLogA problem, which is NP-hard, to the MinIntraBFS problem. The BiMLogA [50] problem is defined as follows: Let $G = (Q \cup D, E)$ be an undirected unweighted bipartite graph with disjoint sets of vertices

Q and D representing query and data vertices. The goal is to find a permutation, π, of data vertices, D, so that the following objective is minimized:

$$\sum_{q \in Q} \sum_{i=1}^{deg_q - 1} \log(\pi(u_{i+1}) - \pi(u_i)) \tag{3.2}$$

where deg_q is the degree of query vertex $q \in Q$, and q's neighbors are $u_1, \ldots, u_{deg_q}$ with $\pi(u_1) < ... < \pi(u_{deg_q})$. Given an instance of BiMLogA, create a new graph $G' = (Q \cup D, E')$, where each undirected edge in E becomes a one-way directed edge in E' from Q to D. Notice that solving MinIntraBFS on G' solves BiMLogA on G. This is because when a node in D is the source for a traversal, its out-degree is zero and does not contribute to the cost in C. When a node in Q is the source, each of the traversals is just one level deep with the source at the root level and its neighbours in D at the next. Since the cost looks at gaps between adjacent nodes in a level, the nodes from Q do not contribute to the cost as they are isolated in their respective root levels. Thus, the final cost is the same as the cost of BiMLogA for G. Since the nodes from Q do not appear in the cost, they can be moved away to one side in the permutation without increasing the total cost. Thus, we have solved BiMLogA for G, which proves the claim of the theorem. □

The identity cost version can be proved similarly. We can also reduce in the other direction. That is, we can go from MinIntraBFS to BiMLogA. We pursue this approach in more detail in Sec. 5.2, where we create an ordering method, BFS-BP, that optimises the MinIntraBFS cost function. BFS-BP is based on the recursive bisection algorithm, BP, proposed by Dhulipala et al. [50].

The second desirable property of reducing gaps *across* BFS levels is more challenging to formulate succinctly. For two successive levels of sizes $|p|$ and $|q|$, we get $|p||q|$ cross terms for pairwise gaps between nodes in these levels. Further, we may need to account for gaps beyond just successive levels. Accounting for inter-level gaps would lead to an explosion of terms. We do not pursue inter-level gaps further in this work. The actual cost

is also different from a $\log$ or identity cost in practice. The cost function depends on the out-degree distribution in the graph as well as the memory hierachy's characteristics. This is also where the UVM hierarchy differs from, for example, cache and DRAM. The unit of data transfers in UVM is a page (4 KB). If two nodes are relatively close in their labels, their neighbourhoods in `elist` may fall on the same page. On the other hand, they may not fall on the same cache line. The cost does not grow proportionately with the gap. Rather, it behaves like a step function. Once we exceed the page boundary, two neighbourhood accesses will fall on two pages, no matter how far the labels end up being. Despite these simplifications, we show in Sec. 6.2 that the MinIntraBFS (log cost) model correlates well with overall performance and data transfer volume. Given the large granularity of transfers, we believe that our ordering methods would also benefit other similar architectures (e.g., non-volatile or solid state storage).

The main reordering algorithm proposed in this work, HALO (Harmonic Locality Ordering) (Sec. 5.1), uses a geometric measure for ordering instead of optimising the cost function directly. Nevertheless, it reduces the MinIntraBFS cost as well as application runtime (Sec. 6). The other proposed method, BFS-BP (Sec. 5.2), is based on optimising the cost function directly. All the reordering methods considered in this work are offline methods. Some prior works treat reordering as an online step [51], but their primary workloads have higher complexity than BFS. Any reordering scheme that scans the graph ends up being as expensive as BFS asymptotically making any proposal for online reordering rather difficult. An ideal reordering scheme in the context of this problem should strive to achieve similar costs as BFS to be practical. Before we describe our proposed solutions, we review related work in the area and also look at a few key metrics from our profiling experiments to better understand the inefficiencies in the UVM model.

Table 3.1: Related Work

Reordering Method	Use Case
RCM [7, 8]	Bandwidth Minimisation
Shingle [49], Slashburn [52], BP [50], LLP [53], BFS [54]	Graph Compression
Gorder [9], Norder [55], RabbitOrder [51], GRO [56], Degree Based [57, 58, 59], EmptyHeaded [60]	Locality Optimisation
METIS [61]	Graph Partitioning

3.2 Related Work

There is a large body of work on graph reordering that spans sparse matrix optimisations, locality optimisations, graph partitioning, and graph compression. We list some of these in Table 3.1. Many sparse matrix computations benefit from reordering. The goal of reordering in these cases is to reduce fill-in in direct solvers or to improve the locality in iterative solvers. Since BFS can be expressed as SpMV, it seems natural to apply similar techniques. The Cuthill-McKee (CM) algorithm [7] and the related reverse Cuthill-McKee (RCM) algorithm [8] are commonly used for reducing the bandwidth of symmetric sparse matrices. RCM is also interesting because the ordering method itself does a variation of BFS traversal. Wei et al. [9] make the observation that sibling relationships between vertices are crucial to locality, and they propose Gorder to reduce CPU cache misses. Gorder's performance for BFS was comparable to RCM in their evaluations. We found that Gorder has a high runtime cost, and we were not able to use it for graphs at our scale. Norder [55] uses (in-)degree + neighbourhood for ordering. As we showed in Sec. 5.1.5, degree can be treated as a first order approximation of harmonic centrality. Norder also looks at neighbours two hops away rather than just immediate neighbours. However, any depth greater than one increases the reordering time asymptotically. A few other schemes also use variations of degree based sorting and clustering [57, 58, 56, 59]. Empty-Headed [60] uses

a hybrid BFS and degree based sorting that is similar to RCM. Rabbit Order [51] is a reordering scheme based on community detection for page-rank like applications. It works on undirected graphs, and the performance for BFS is comparable to RCM in the authors' results. We found that a lot of reordering methods target the locality of vertex property structures (i.e., the vector in SpMV like formulation). These methods help in improving the temporal locality of properties such as page-rank or distance values of nodes. They do not benefit accesses in `elist` in our semi-external memory model. There is no temporal reuse of edges in a BFS. Various graph compression methods [49, 52, 50, 53, 54] help with locality in general, although the goals are somewhat different. Graph partitioning methods such as the ones used by METIS [61] are also employed in distributed computing scenarios. The goal in partitioning is to reduce the communication costs and to improve load imbalance. This does not map to our model directly as there is only a single compute node that cannot keep a partition in memory across iterations. There are also disk based methods like GraphChi [2], which uses a sharding technique for improving locality. These methods also work better for iterative convergent algorithms like page-rank that need to access the entire edge list in each iteration.

CHAPTER 4

UVM PERFORMANCE CHARACTERISTICS

4.1 Read Amplification

We compare the volume of data transferred over the interconnect (i.e., from the host to the GPU) to the ideal volume that needs to be transferred for doing a breadth first traversal. The ideal volume is the volume of data that would be transferred if the GPU's memory were large enough to hold the data and the unit of transfer were a single edge. In other words, it is the number of edges traversed times the unit edge size. The ideal transfer size is upper bounded by the graph size since the BFS can traverse each edge at most once, although it can be lower if the traversal doesn't touch the entire graph. This can be the case for directed graphs or undirected graphs with multiple connected components. We show the results in Fig. 4.1, where we see that on average we transfer 2.3x the volume of data when we do traversals in the UVM model.

4.2 Prefetches

A portion of the total data transfer volume can be ascribed to prefetches. Traditional cache-line prefetchers perform poorly in prefetching dependent irregular access patterns such as those seen in graph traversals. However, the prefetcher in a unified memory system operates under different conditions. The interconnect has very high latency, and small transfers are extremely inefficient. Hence, GPU page faults are handled in batches, which are called page fault groups in NVIDIA's terminology [36, 37, 38, 39]. When the GPU raises a page fault exception, the driver handles the pending page faults in the group. While faults are being handled, the GPU is not allowed to raise more exceptions. Instead, it pushes the faults to a buffer. Once the current fault group is serviced, the GPU is allowed to raise an

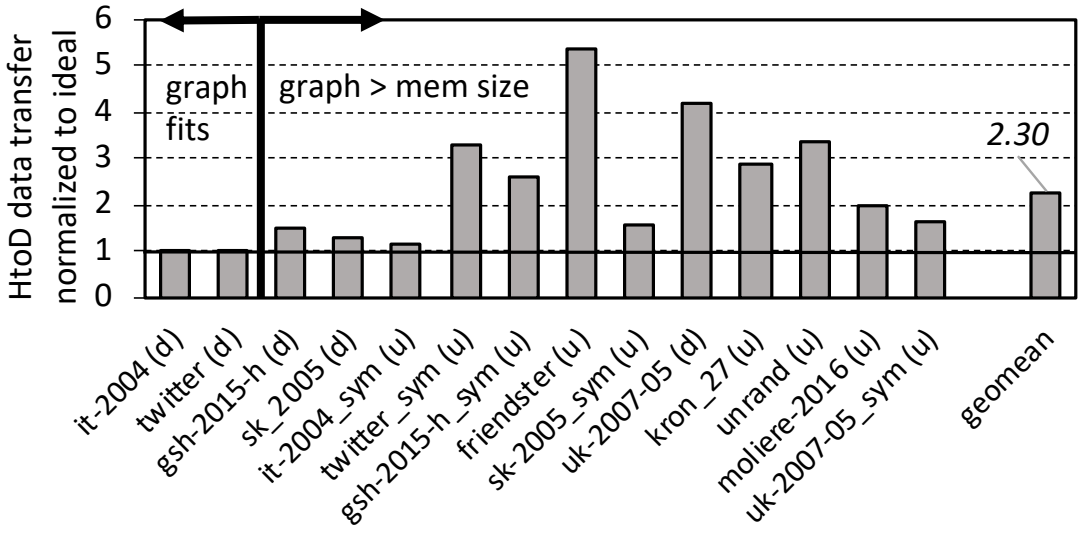


Figure 4.1: Read amplification for BFS on a Titan Xp GPU

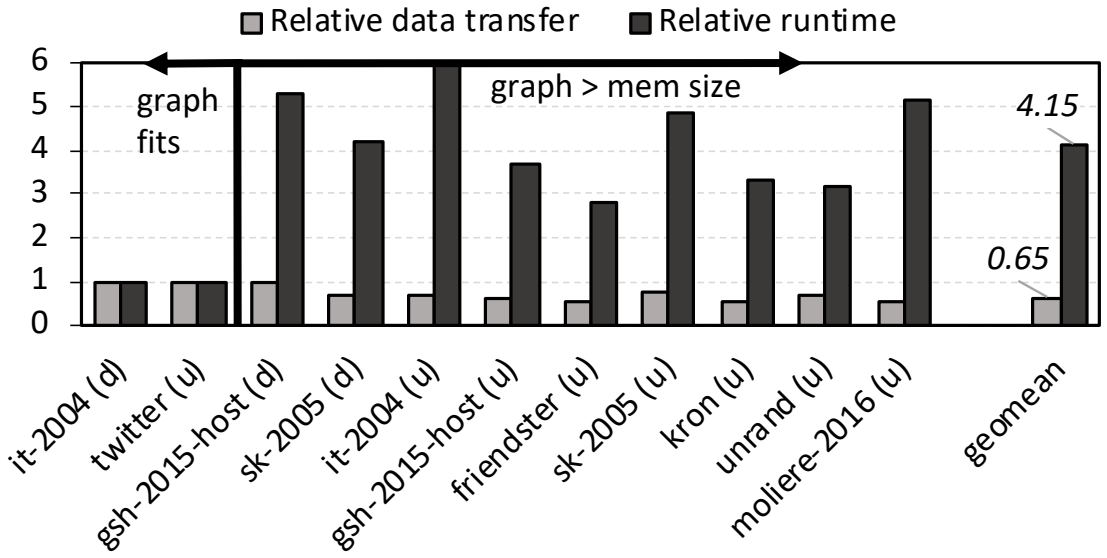


Figure 4.2: Data transfer and runtime metrics for BFS with prefetch disabled for Titan Xp relative to those with prefetch enabled. Despite a reduction in data transfer, performance suffers severely

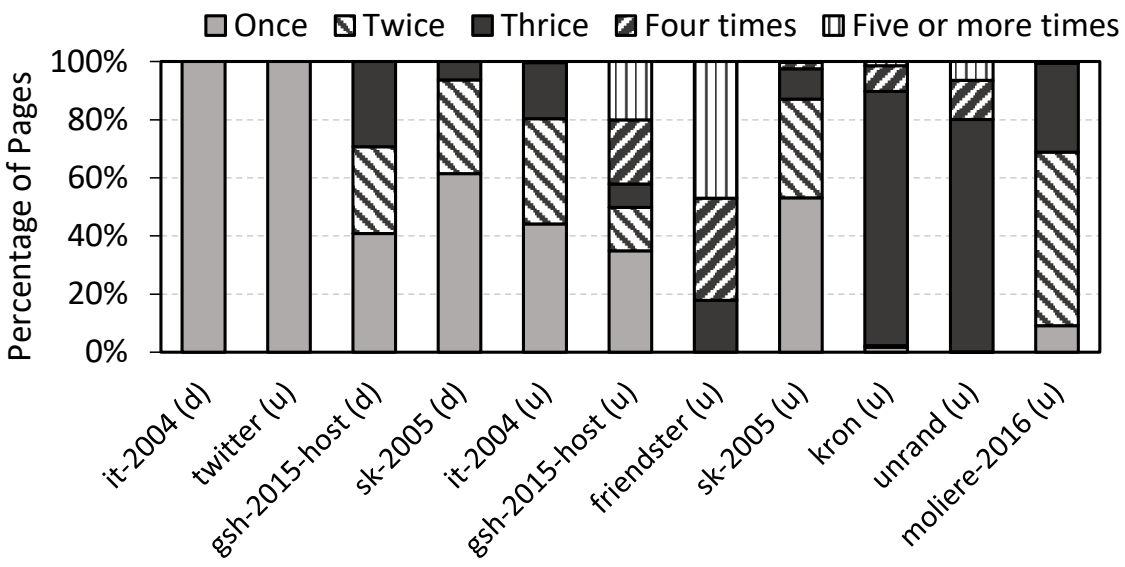


Figure 4.3: Distribution of the number of times the same pages are fetched from host to device due to evictions. Note that the first two graphs fit in memory.

exception again which starts the processing of the next group. The prefetcher uses heuristics to transfer more pages than requested based on the faulting addresses of the pages in the group. We see that disabling the prefetcher in an attempt to reduce the amount of data transferred hurts performance severely. In Fig. 4.2, we see that disabling the prefetcher reduces the data transfer size to 0.65x the baseline on average. However, it increases the total runtime to 4.15x times the baseline.

4.3 Evictions

The poor locality of accesses in graph applications also makes page replacement decisions challenging. In the current implementation for NVIDIA GPUs, evictions happen at a 2 MB granularity. When a new page needs to be mapped in, a 2 MB victim region is selected for eviction, and any 4 KB pages mapped in the region are evicted. The eviction follows an age based LRU policy. It is different from traditional LRU in that it tracks the timing of page allocations rather than page accesses by the GPU. When the GPU requests a page from the host due to a GPU page fault, the host promotes its corresponding 2 MB region to the MRU

position. Conversely, the pages in the 2 MB region from the LRU position are evicted. We confirmed this high level mechanism from the UVM driver's source code. In Fig. 4.3, we plot the distribution of 4KB pages that are fetched from host to device for the baseline UVM based BFS implementation from Fig. 2.1. We see that it is extremely common for a large fraction of pages to be fetched multiple times. These pages are originally fetched as either demand loads or prefetches, but are evicted due to capacity constraints, only to be fetched again in the future.

CHAPTER 5
GRAPH REORDERING METHODS

5.1 Harmonic Locality Ordering (HALO)

There are two primary challenges with reordering a graph for efficient accesses in an arbitrary BFS traversal: i) The source node for BFS is not known beforehand, and ii) Directed graphs make reordering challenging since the locality of accesses is highly dependent on the directionality of edges. If the source vertex for the BFS traversal were known beforehand, one could label the vertices in the graph in the order in which they are actually visited during the BFS starting with the source as vertex 0. The sample graph in Fig. 2.2 is in fact labelled in BFS order from node 0. Hence, it is not surprising that it has good locality of accesses in `elist` for that particular traversal. As we discussed in section 2.3, this does not extend to other traversals. Our main contribution is that we devise effective and efficient heuristics based on centrality scores to create an ordering. We describe our solution in stages as we refine it towards the final ordering algorithm.

5.1.1 Connected Undirected Graphs

For simplicity, let us consider undirected graphs with a single connected component and ignore any costs of reordering for the time being. We would like to place the nodes in BFS frontiers close to each other in their ordering, but we do not know the source beforehand. Instead of a particular source, let us consider all the sources as the cost in equation 3.1 suggests. If we do a BFS from every source, we would know when a node is visited in each of these traversals. Some nodes are likely to be discovered sooner than the others on average if we look at all possible sources. If we were to create this distribution, we can use this heuristic to order the graph starting with the node that is most likely to be discovered.

Formally, for each node, we can compute

$$c(x) = \frac{1}{\sum_{y \neq x} d(y, x)} \tag{5.1}$$

where $d(y, x)$ is the shortest distance from a different node y to x. This is, in fact, a centrality metric known as closeness centrality [62, 63] that is used in network and social analysis. A node with high closeness centrality is close to all other nodes in the graph since it has a lower average distance from other nodes. In other words, it is likely to be discovered soon in an arbitrary BFS. The node with the highest $c(x)$ can be ordered first followed by the next highest $c(x)$ and so on in this ordering scheme. The interesting relation is that we can go from a spatial metric in the graph (i.e, distance of a vertex from others) to temporal aspects of the traversal algorithm (i.e., when a vertex is likely to be in a frontier) back to spatial aspects of the graph's storage (i.e., how to lay out the node labels).

5.1.2 Efficiency Considerations

Computing the closeness centrality in unweighted graphs reduces to performing a BFS from each node, which amounts to $\mathcal{O}(n(n + m))$. As a preprocessing step, this is impractical. Instead, we can sample the starting nodes. If we choose k nodes uniformly at random as starting nodes, and use these samples to compute the closeness centrality values, Eppstein et al. [64] show that for a graph with diameter d and an error term ϵ, the inverse centrality can be approximated to within an additive error of ϵd if k is $\Theta(\log n/\epsilon^2)$. Since real world graphs such as social networks have a small diameter [65], the approximation works well in practice for such cases. The total complexity of this scheme in unweighted graphs is the cost of doing BFS from each of the sampled nodes, which gives us $\mathcal{O}((n+m)\log n/\epsilon^2)$. Further, each of these sampling BFSes can be done in parallel since they are independent. This is in addition to the parallelism in each individual BFS.

5.1.3 Directed and Disconnected Graphs

Notice that closeness centrality is only defined for undirected and connected graphs. Since distances are summed, directionality as well disconnectedness introduces infinities in the mix. For example, consider a node that can only be reached from a few nodes. The distance of such a node from all other nodes is infinite, and trying to compute the closeness centrality from equation 5.1 would collapse the entire term. Instead of summing distances, we sum the reciprocal of distances. Formally, we compute

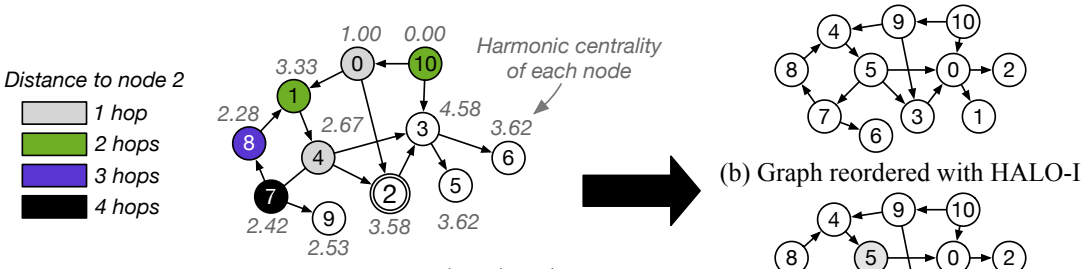
$$H(x) = \sum_{y \neq x} \frac{1}{d(y, x)} \tag{5.2}$$

While this looks similar to 5.1, notice that taking the reciprocal of distances before summing them makes it a harmonic sum rather than an arithmetic one. This deals with the problem of infinite distances elegantly as it doesn't collapse the term. This centrality metric has been termed as harmonic centrality [66, 67, 68, 69] and is relatively new compared to the previous closeness metric. We can use the same sampling based approximation method from closeness centrality, and Eppstein et al.'s bounds still apply [70]. The first variant of our ordering scheme, called **HALO-I** hereafter, orders the nodes of a graph in descending order of their harmonic centrality scores. The algorithm for calculating approximate harmonic centralities is presented in Alg. 2. In Fig. 5.1, we show how the original sample graph from Fig. 2.2 is permuted by this algorithm. The centrality scores are denoted outside the nodes with explicit calculation shown for node 2. Node 3 is the one with the highest harmonic centrality, followed by nodes 5 and 6. Hence they get labelled as nodes 0-2 in the reordered graph. The rationale is that node 3 in the original graph is central for an arbitrary BFS and likely to be discovered soon.

Algorithm 2 HALO-I (Approximate Harmonic Centrality)

Require: $G = (V, E)$, a sample parameter **k** > 1
Ensure: harm[n] array of size n with centrality scores

```
for i = 0 to k do
    source[i] = get_random_vertex()
    levels[i][] = BFS(G, source[i])
end for
for j = 0 to |V| do
    harm[j] = 0
    for i = 0 to k  do
        // Skip zero level values (source nodes)
        if (level[i][j] != 0) then
            harm[j] = harm[j] + 1/(level[i][j])
        end if
    end for
end for
for i = 0 to k do
    // Scale values for source nodes that were skipped
    harm[source[i]] = (k * harm[source[i]] ) / (k-1)
end for
```



Harmonic centrality of node $2 = 2 \times 1 + 2 \times \frac{1}{2} + \frac{1}{3} + \frac{1}{4} = 3.58$

(a) Original graph with harmonic centralities

(b) Graph reordered with HALO-I

(c) Graph reordered with HALO-II

Figure 5.1: Graph reordering with HALO. HALO-I orders in descending order of centralities. HALO-II additionally orders the neighbours of nodes before moving to the next centrality score

5.1.4 Biasing with Neighbourhood

While HALO-I gives us centrality scores, these scores can be the same or very close for nodes that may be in different parts of the graph. Since our primary problem is to optimise locality for traversals, we make a modification where we bias the labelling to favour (out-)neighbours of nodes that we think are likely to be discovered soon. The modification is that instead of going in strict descending order of centralities for labelling, at each step we also try to label the neighbourhood of nodes wherever possible. This variant is described in Alg. 3 and called **HALO-II** hereafter. The ordering becomes a nested loop where the outer one (line 2) goes in centrality order, and the inner one (line 6) labels the unlabelled neighbours of nodes. In this scheme, some such neighbours may already have been labelled earlier, either because they have higher centralities themselves, or because they are neighbours of nodes with higher centralities. Such nodes would be skipped. We also show how this changes the ordering in our running example in Fig. 5.1. When node 4 in the original graph is labelled as node 5, HALO-II labels its neighbour (node 7 in the original graph) next. Hence, labels 6 and 7 are swapped in the two variants. This method is still making a single pass over the graph where each edge is visited only once. So it does not increase the cost asymptotically.

Algorithm 3 HALO-II (Biasing with Neighbourhood)

Require: $G = (V, E)$, harm[n] calculated in Alg. 2
Ensure: A bijection $\phi\colon V \to V$

```
 1: count = 0
 2: for all u ∈ V in descending order of centrality scores do
 3:     if φ[u] not valid then
 4:         φ[u] = count++
 5:     end if
 6:     for all v ∈ V such that (u,v) ∈ E do
 7:         if φ[v] not valid then
 8:             φ[v] = count++
 9:         end if
10:     end for
11: end for
```

5.1.5 Degree Based Ordering as a Special Case

We described the process of approximating harmonic centrality by sampling BFSes in Alg. 2. It is possible to avoid this step with a trade-off in accuracy. We can look at harmonic centrality of a node in equation 5.2 as sum of terms with a decay function. The higher order terms have their weights discounted by a factor equal to the distance they are away from the node [71]. The first term in the harmonic centrality of a node is the number of nodes that are at distance one, which is its (in-)degree. The second term is the number of nodes at distance two, but this term is scaled by the factor 2. The third term is scaled by a factor 3, and so on. This means that we can get a first order approximation of the centrality from its (in-)degree, and we can replace the step of sampling BFSes with this approximation. We evaluate these simplifications for both the variants of our ordering algorithms in Chapter. 6.

5.2 Recursive Graph Bisection (BFS-BP)

We create an additional ordering method, BFS-BP, based on the optimisation of the objective function in the MinIntraBFS problem from Chapter 3. Recall that the objective in MinIntraBFS is:

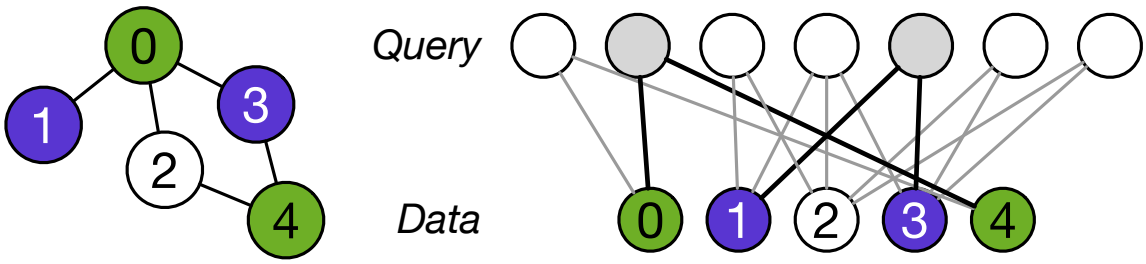
$$C = \sum_{\text{sources}} \sum_{\text{levels}} \sum_{i=1}^{|\text{level}|-1} c(\pi(u_{i+1}) - \pi(u_i)) \tag{5.3}$$

The idea is to reduce MinIntraBFS to the BiMLogA problem, and leverage the recursive bisection algorithm, BP [50], for BiMLogA. Note that we are using log as the cost function c in this reduction. The BiMLogA formulation seeks to reduce gaps between the data nodes (D) in a bipartite graph consisting of query (Q) and data nodes (D). BiMLogA seeks to minimise the following objective:

$$\sum_{q \in Q} \sum_{i=1}^{deg_q - 1} \log(\pi(u_{i+1}) - \pi(u_i)) \tag{5.4}$$

To use this formulation, we add a query node for each level of a BFS from each source,

and we add edges from the query node to the nodes in the respective BFS level. In Fig. 5.2, we show a sample graph and the corresponding BiMLogA graph that we need to create in order to optimise the objective in MinIntraBFS.



(a) Sample graph (b) Reduction to BIMLogA graph

Figure 5.2: Reduction from MinIntraBFS to BiMLogA. When node 2 is the source, {0,4} and {1,3} are the level sets. We add one query node for each set along with the corresponding edges to the data nodes. The same process is followed for a BFS from each source

We outline the main aspects of the algorithm here, and refer the reader to BP [50, 72] for more details. The algorithm follows the Kernighan-Lin [73] heuristic for recursive graph bisection. The set D of data nodes is split into into two sets, $V1$ and $V2$, and a computational cost of the partition is defined. Next, the algorithm exchanges pairs of vertices in $V1$ and $V2$ in order to improve the cost. For every vertex $v \in D$, a move gain, which is the difference of the cost after moving v from its current set to another one, is computed. Then the vertices of $V1$ and $V2$ are sorted in the decreasing order of the gains to produce lists $S1$ and $S2$. The lists $S1$ and $S2$ are traversed in order and pairs of vertices are exchanged if the sum of their move gains is positive. This describes a single iteration, and the same step continues for a fixed number of iterations or until a convergence criterion is met. The algorithm continues recursively on the newly created partitions. The cost of the partition, which guides the move gains in BP, is defined as follows: For every vertex $q \in Q$, let $deg_1(q) = |(q, v) : v \in V1|$, that is, the number of adjacent vertices in set $V1$; define $deg_2(q)$ similarly. Then the cost of the partition is:

$$\sum_{q \in Q} \left(deg_1(q) \log(\frac{n_1}{deg_1(q) + 1}) + deg_2(q) \log(\frac{n_2}{deg_2(q) + 1}) \right) \tag{5.5}$$

where n_1 is $|V_1|$ and n_2 is $|V_2|$. The cost estimates the number of bits needed for coding the gaps between vertex labels using binary coding. This follows from the fact that if the neighbours of $q \in Q$ are uniformly distributed in the final arrangement of $V1$ and $V2$, then the the average gap between consecutive numbers in the q's adjacency list is $gap_1 := n_1/(deg_1(q) + 1)$ and $gap_2 := n_2/(deg_2(q) + 1)$ for $V1$ and $V2$ respectively.

The cost of running BP on a graph is $\mathcal{O}(m \log n + n \log^2 n)$ [50]. However, in order to create the BiMLogA graph, we need to do a BFS from every source, which is $\mathcal{O}(n(n+m))$, and the resultant BiMLogA graph would have $\mathcal{O}(n^2)$ edges since we add $\mathcal{O}(n)$ edges for each BFS. This is again not practical as a preprocessing step. Similar to sampling strategy in HALO, we only do BFSes from the sampled nodes instead of all nodes. This method is called **BFS-BP** hereafter.

CHAPTER 6
GRAPH REORDERING RESULTS

We compare the performance impact of different ordering schemes in two different experiments. From the ordering methods in prior works, we use RCM and BP in our evaluations. RCM is either the best, or close to the best performing ordering for BFS in prior works [9, 51], and BP is competitive for graph compression metrics. RCM is used for reducing the bandwidth of sparse symmetric matrices (i.e., undirected graphs). For directed graphs, we use RCM on the symmetrised version (i.e., $A + A^T$) of the graph. In the first experiment, as shown in Fig. 6.1, we look at the performance of 50 different traversals performed one at a time from random sources. In Fig. 6.2, we also look at the performance of multi-source BFS traversals where each kernel does 10 different traversals from random sources at the same time. This is meant to simulate patterns seen in applications like betweenness centrality [14, 74], all pairs shortest paths, or BFS in large diameter graphs [75], where instead of a single frontier, multiple frontiers progress at the same time. The reordering methods compared are natural ordering (baseline), random ordering, the two HALO variants and their first order approximations to (in-)degree, BP [50], RCM [7], and BFS-BP. For the sampling based approximation in HALO and BFS-BP, we used 20 BFS traversals from different and unrelated random sources.

6.1 Speedup

We first discuss the results in Fig. 6.1. The ordering methods are arranged in increasing order of average performance. The baseline's ordering is natural ordering. There is often some locality in natural ordering, which depends on the process used for generating the graph. For example, the web graphs in this collection are ordered based on the lexicographical ordering of URLs. This is important because the potential for improving locality

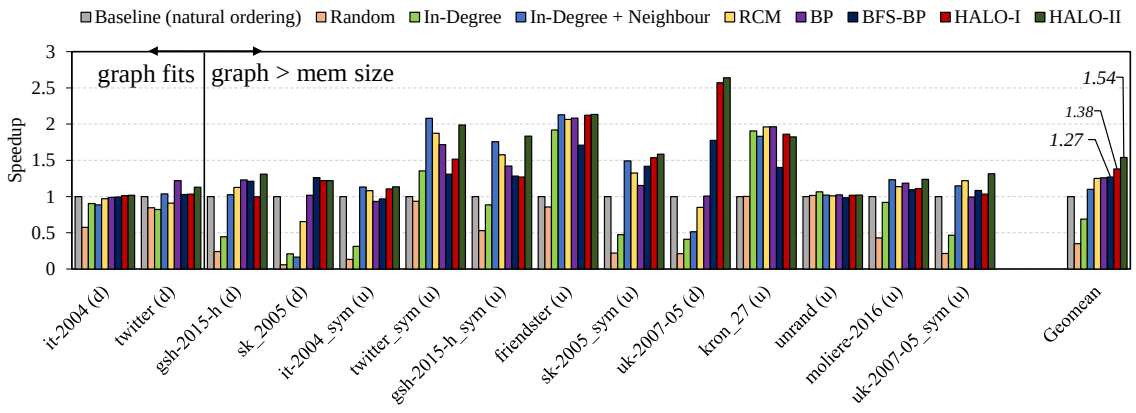


Figure 6.1: Single-source BFS performance for different graph orderings relative to natural ordering on a Titan Xp GPU. Results are accumulated over 50 traversals from random sources. Geomean reported for graphs larger than memory capacity.

is highly dependent on the original ordering. The two synthetic graphs, `kron_27` and `unrand`, don't have a natural ordering, and their baseline is the same as random ordering. Since `unrand` is a uniformly random graph, it does not benefit from any ordering scheme. The random ordering destroys any existing locality in other graphs and leads to an average slowdown of 65%. The simple in-degree based ordering scheme also leads to an average 31% slowdown. However, with the addition of immediate neighbour heuristic to the in-degree scheme, the performance improves, but only for undirected graphs. It performs poorly for directed graphs, and the overall average speedup is 1.1x. Next, RCM and BP perform better than in-degree + neighbour, but the performance for directed graphs is again variable with better results for undirected graphs. Overall, RCM and BP show speedups of 1.25x and 1.26x, respectively. BFS-BP performs more consistently across both directed and undirected graphs, but does not outperform RCM for undirected graphs. This is consistent with prior works which find RCM to perform well for undirected graphs. BFS-BP's overall speedup is 1.27x. Finally, HALO-I and HALO-II show an overall speedup of 1.38x and 1.54x, respectively. BP, BFS-BP, and HALO are consistent across the datasets in that they perform at least as well as the baseline for every case. Since the original graph may already have good locality, the reordering method should not destroy it. The performance

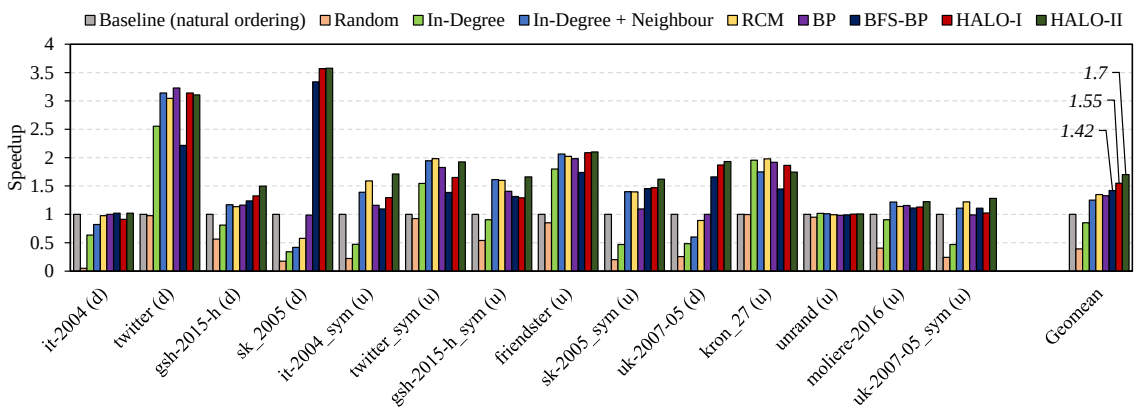


Figure 6.2: Multi-source BFS performance for different graph orderings relative to natural ordering on a Titan Xp GPU. Each kernel does a BFS from 10 random sources at the same time.

trends for the multi-source BFS experiment in Fig. 6.2 are similar with BFS-BP, HALO-I, and HALO-II achieving speedups of 1.42x, 1.55x, and 1.7x, respectively. There are two graphs at the left end that exhibit larger swings in performance. The `twitter_d` graph fits in memory on its own, but with the additional metadata needed for storing BFS levels for multiple sources, it starts to exceed the GPU's memory. Reordering the graph manages to keep the accesses largely in-memory. This leads to a more pronounced speedup. This effect diminishes once the graphs become larger, and the overall trend becomes similar to Fig. 6.1.

To demonstrate the results visually, we show the distribution of active nodes with the passage of time in Fig. 6.3. The vertical axis plots increasing level numbers from top to bottom, and the horizontal axis plots the vertex ids active at each level. For the directed uk web graph, we can clearly see that the active nodes *within* the levels as well as *across* the levels follow the desirable "staircase" pattern when the graph is ordered with HALO-II. BFS-BP on the other hand manages to reduce the overall spread of active vertices within levels, as evidenced by the gap at the right end, but does not outperform HALO-II. RCM fails to improve over baseline for the directed graph. For the undirected friendster graph, both RCM and HALO-II achieve good results whereas BFS-BP falls behind the others.

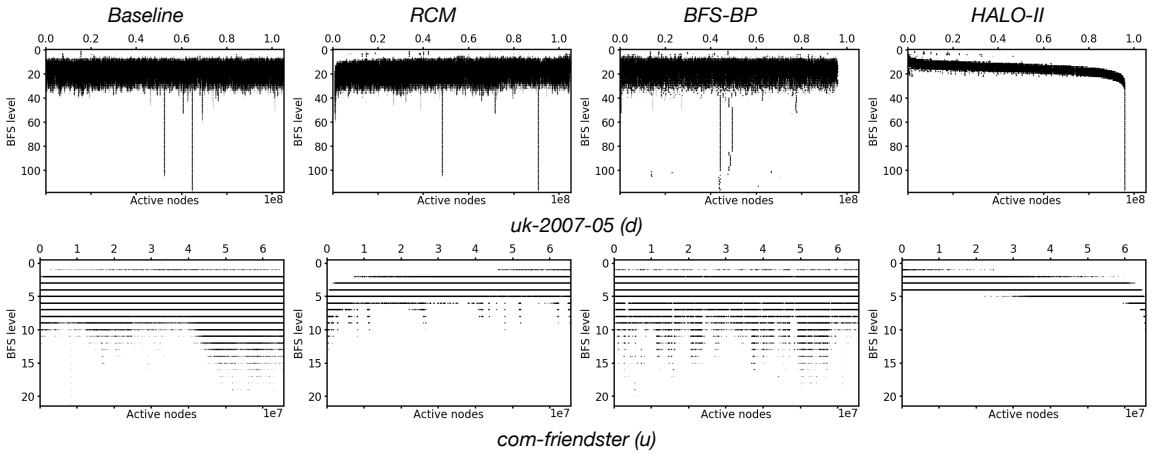


Figure 6.3: Distribution of active nodes across different frontiers of the traversal

6.2 Data Transfer Volume and Log Gap Cost

The results show that the performance improvements due to reordering are strongly correlated with the reduction in data transfer volume. Earlier in Sec. 4, we observed that the baseline ordering causes read amplification of 2.31x over the expected ideal transfer volume. In Fig. 6.4, we see that HALO-II reduces this amplification from 2.31x to 1.42x. BFS-BP and RCM manage to reduce it to 1.72x and 1.8x, respectively. We also compare the average log gap between nodes within a BFS level in Fig. 6.5. This is the MinItraBFS cost from Sec. 3, Eq. 3.1 for the evaluated traversals. Once again, we see that HALO-II achieves the lowest average log gap cost, which is consistent with the volume of data transfer in Fig. 6.4 as well as performance trends in Fig. 6.1. The degree of improvement in the log gap cost is also a good indicator of improvement in performance. We see that graphs like the `uk-2007-05(d)` show a significant reduction in the log gap cost, but the cost is relatively unchanged in `unrand`. Although BFS-BP is designed to minimise the log gap cost explicitly, it is still an approximate solution with simplifying assumptions. For example, the cost of the partition in Eq. 5.5 assumes that neighbours of $q \in Q$ are uniformly distributed in the final arrangement. We also using sampling as an additional simplification to limit the runtime.

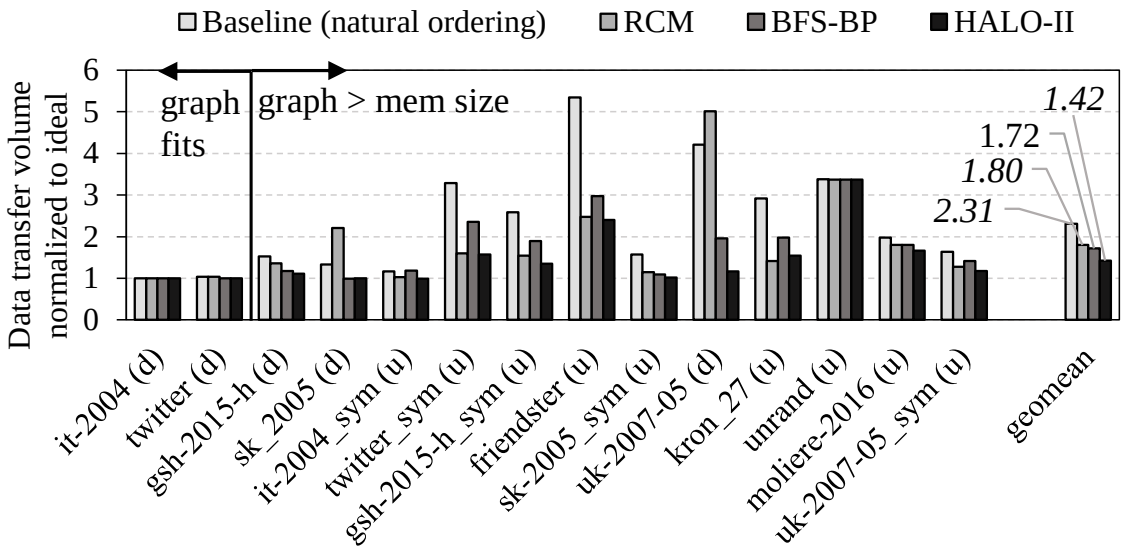


Figure 6.4: Host to device data transfer volume (lower is better) relative to ideal transfer volume

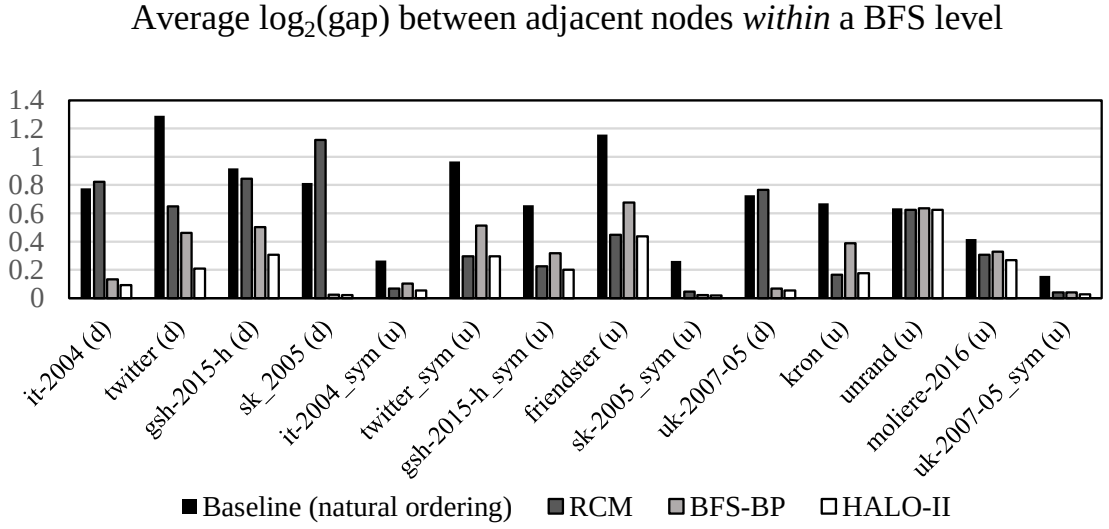


Figure 6.5: Average log gap (i.e., MinIntraBFS cost; lower is better) between nodes within a BFS level for 50 traversals

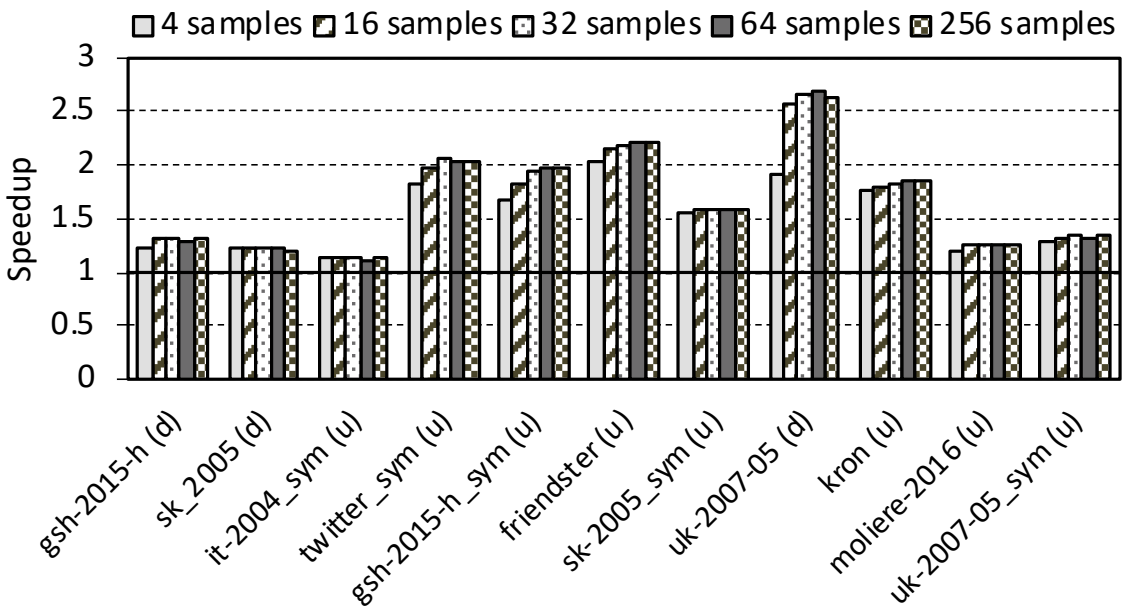


Figure 6.6: Sensitivity to Number of Samples for HALO-II

6.3 Sample Size Sensitivity and Reordering Overhead

The practicality of using harmonic centrality as a tool for graph reordering hinges on the number of sample BFS traversals that are needed for approximating it. For the experiments in Fig. 6.1, we used 20 random samples. We now vary the sample parameter systematically from 2 to 256 and measure the performance of BFS on reordered graphs in Fig. 6.6. The result show that performance saturates around 32 samples for these graphs with negligible improvement beyond that. These graphs are in the 100M node range (Section 2.4). Eppstein et al.'s bounds [64] on approximate closeness centrality and the extension to harmonic centralities [70] show $\Theta(\log n/\epsilon^2)$ samples as sufficient for an additive error bound of ϵd. From our sampling sensitivity results, they saturate in performance much sooner than the bounds (i.e., more like $\log n$ rather than $\Theta(\log n/\epsilon^2)$). There are a few likely reasons for this. One is that we are using centralities indirectly for ordering. Specifically, we are using a relative ordering of centralities. So the absolute error in centrality values is less relevant for our case. Further, since UVM is a page fault handling mechanism, we benefit

from locality improvements at a coarse granularity and are more tolerant to errors at a finer level.

Sequential Costs: The cost of both the variants of HALO is asymptotically $\mathcal{O}(\log n/\epsilon^2(n+m))$ since sorting centralities is $\mathcal{O}(n \log n)$ and the second variant in Alg. 3 is just a scan over the graph ($\mathcal{O}(n+m)$). The cost of RCM is $\mathcal{O}(m)$ [76]. This does not include the cost of finding pseudo peripheral nodes, which is an essential additional step in RCM. We are not aware of bounds on finding pseudo peripheral nodes, but various methods [77, 78, 79] have been proposed for pruning the number of additional BFSes needed. BFS-BP costs $\mathcal{O}(kn \log n + n \log^2 n)$ if we use k sample BFSes since each BFS adds $\mathcal{O}(n)$ edges.

Parallelisation: The computation of approximate harmonic centralities is trivially parallel since the sample BFSes are completely independent and each BFS can also be parallelised. In contrast, the additional BFSes needed for finding pseudo peripheral nodes for RCM are dependent in nature. Parallelising RCM itself is also challenging [80]. We note that the additional pass over the graph in HALO-II (Alg. 3) is sequential in our current implementation. BFS-BP follows recursive parallelism where each partition becomes an independent subtask, but processing within a partition is still serial.

Runtime: Since we did not have uniformly optimised and parallel versions of the different ordering methods, we do not compare the raw runtimes. We used the boost library's implementation for RCM, which was slow and took several hours. We used the PISA framework [72, 81] for BP. It uses TBB for parallelization, and solving BFS-BP with 20 BFS samples took roughly one hour for the graphs in this work on a 44 core Broadwell server. Reordering with HALO-I and HALO-II generally took minutes to tens of minutes.

6.4 Scaling with Graph500 Graphs

We perform scaling experiments with Kronecker graphs that are generated per Graph500 specifications using the gapbs framework [82]. These graphs have 2^{scale} nodes and $m = 16 \times n$. We perform experiments up to scale 30 for the directed case and up to scale 29 for

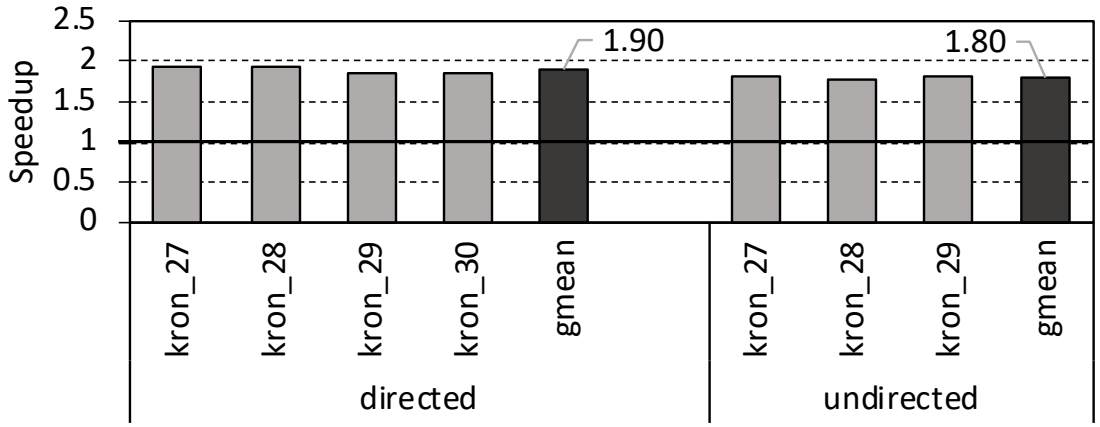


Figure 6.7: Scaling with Kronecker graphs. Number denotes the scale where $n = 2^{scale}$ and $m = 16 \times n$

the undirected case since we hit our system's host memory limit (256 GB) at that scale. We see a consistent speed-up of around 1.8-1.9x for all the cases.

6.5 Additional Optimisations

In the current implementation for UVM in NVIDIA GPUs, the pages allocated with UVM only exist in a single device's page table at a time. All memory is assumed to be read-write (RW) conservatively. The runtime does not have any transparent fine-grained copy-on-write mechanisms. When RW pages need to be evicted from the GPU, they need to be transferred and mapped in the host's page table. This is a blocking operation which effectively doubles the data transfer volume since everything that is transferred from host to device is also transferred back for a large enough data set due to evictions. Our application is clearly segregated between read only (RO) structures: `elist` and `vlist`, and RW structures `vprop` for storing level numbers. This allows up to pass a read-only memadvise hint for the RO structures. The GPU create copies of such pages and drops them freely on eviction instead of waiting for them to be transferred back to the host. This leads to an overall speedup of 1.85x due to the approximate halving of data moved over PCI-e. While this is a simple optimisation, it may not be immediately obvious or possible in all

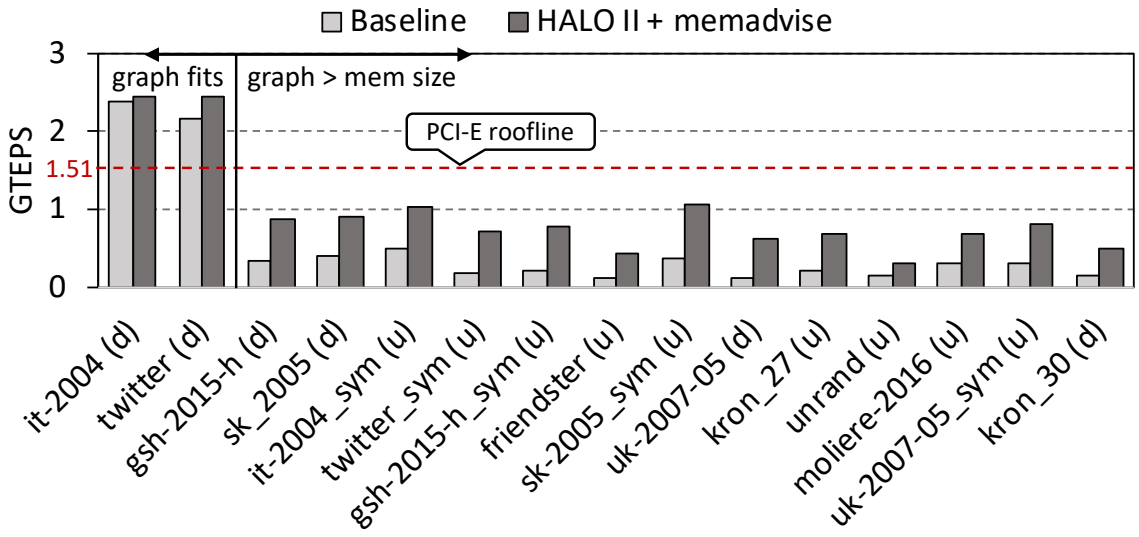


Figure 6.8: Overall performance after combining HALO-II reordering and the read-only memadvise hint

applications. Combined with the average speedup of 1.54x from HALO-II's reordering, we get an overall average speedup of 2.84x which we show in Fig. 6.8.

6.6 Limitations and Discussion

Although we did not encounter pathological cases in our datasets, it is possible to create graphs where harmonic centrality will not create a good ordering for BFS. For example, consider multiple identical copies of a graph as different (weakly) connected components. Since the components are identical in a geometric sense, the centrality values will be identical for every copy of a node in each of the components. This will create an interleaved ordering of nodes if we sort by centrality scores. In this particular case, we can separate the components first before ordering, but in general, one can create similar scenarios even within a single component.

The semi-external memory model, where we assume that only $\mathcal{O}(n)$ structures fit in memory, also places some restrictions on algorithms. For example, some BFS implementations allow duplicates in the frontier and remove duplicates in a subsequent step. This

strategy does not work in the UVM model as the frontier would grow to $\mathcal{O}(m)$ with duplicates and cause a severe performance penalty due to random read-write traffic over the slow interconnect. This does not affect our level array implementation of frontiers that fits in memory, but is a fairly common pattern in graph analytics.

The benefit of graph reordering is also application dependent. In this work, we looked at improving the locality for edge accesses that are hard to predict. In some applications like page-rank, we need to touch all the edges in each iteration. This sort of reordering may not be beneficial, but prior solutions such as GraphChi [2] may apply. In some applications such as single source shortest paths (SSSP), work-optimality considerations are more important that ordering. Common parallel implementations such as Bellman-Ford's algorithm do asymptotically more work than the sequential counterpart, which leads to a severe performance penalty due to remote accesses. Delta-stepping [83, 84] implementations for SSSP have been proposed, but are quite challenging to implement on GPUs. Finally, the UVM architecture is still relatively new and would benefit from a combination of hardware and software solutions. Recent hardware proposals [85, 86] that optimise the GPU's page fault handling mechanism can be used as a complimentary approach.

6.7 Conclusion

In this work, we looked at the problem of improving the locality of data accesses for breadth first search in a system architecture where the GPU can overprovision memory and access it from the host transparently. This poses several challenges as BFS has low computational complexity and an irregular data access pattern. We proposed a new graph reordering algorithm, HALO, that is both lightweight and effective in improving the performance of subsequent BFS traversals on large real world graphs. It is possible to improve the performance of BFS by 1.5x-1.9x in the unified memory setting by reordering the graph. We found that HALO captures the structure of real world directed graphs from the perspective of an arbitrary BFS traversal well whereas prior popular methods such as RCM only do so

for undirected graphs. Additionally, the problem ties into other graph ordering problems, and we showed that we can leverage prior techniques from the graph compression domain such as recursive bisection which resulted in an additional ordering method (BFS-BP). In general, this opens up the problem space for creating orderings that are both locality and compression friendly. To our knowledge, neither the problem of doing graph traversals in unified memory, nor the use of harmonic centrality for graph reordering have been explored before. The reordering solution is general enough that we expect it would extend to other similar memory hierarchies with large unit transfer sizes.

CHAPTER 7
GRAPH COMPRESSION FOR GPUS

In the dissertation so far, we used the compressed sparse row (CSR) format for representing static graphs. While CSR is extremely common in high performance graph applications and sparse linear algebra, alternate representations can compress sparse graphs better. Besta et al. [5] categorise a number of lossless graph compression methods developed over the past few decades. Their taxonomy covers a number of theoretical and practical approaches spanning different domains such as web and social graphs, biological networks, graph databases, etc. From these methods, we draw attention to the WebGraph [42] framework, which is a state of the art CPU based graph compression framework that is used extensively in practice for massive web and social graphs. WebGraph uses a number of techniques including gap encoding, reference encoding, differential encoding, and interval encoding. A number of graph ordering and clustering techniques such as BP [50], LLP [53], SlashBurn [49], can be used in conjunction with the underlying compression scheme. WebGraph often compresses graphs to fewer than 10 bits per edge, which is a substantial saving over 32 or 64 bits in the CSR representation. On one hand, an efficient graph representation can, for example, move a graph from the semi-external memory regime, as in UVM, to the in-memory regime. On the other hand, schemes like WebGraph are difficult to adapt to GPUs directly since decompression can be sequential and branch intensive. Our focus in the second half of the dissertation is to devise a graph representation that is amenable to run-time decompression on GPUs.

7.1 Motivation

When the graphs exceed memory capacity of GPUs, there are different approaches [87, 88, 89] for out-of-core processing. As we saw earlier, baseline UVM suffers from poor

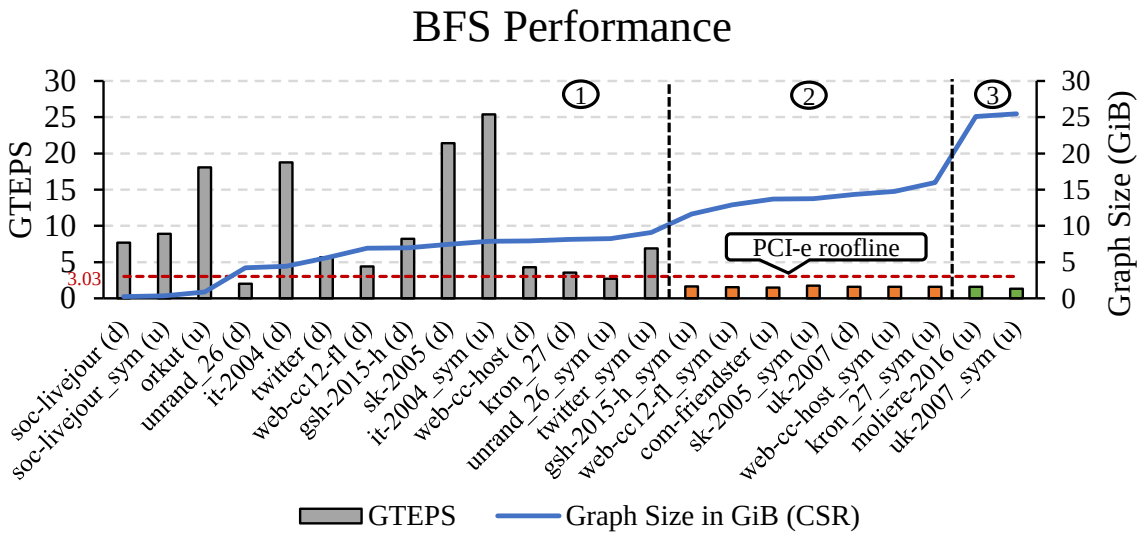


Figure 7.1: BFS performance measured in billions of Traversed Edges per Second (GTEPS) on a Titan Xp GPU with 12 GiB memory. Regions: (1) Graphs that fit in memory. (2) Graphs that exceed memory, but would fit after compression. (3) Graphs that will exceed memory even after compression.

Table 7.1: GPU Bandwidth Characteristics

GPU	Mem.	HtoD Link	DtoD BW	HtoD BW
Titan Xp	12 GiB	PCI-e 3.0	417.4 GB/s	12.1 GB/s

performance due to issues such as I/O amplification, thrashing, and evictions. An alternate approach, as shown in EMOGI [87], is to use *zero-copy* memory transfers instead of UVM where the memory region always resides on the CPU and data is streamed at cacheline granularity. Since the data never resides on the GPU, this avoids issues arising from page-fault handling. This works quite well in cases where there isn't significant reuse of data. The benefit of this approach is that it does not require preprocessing and serves as a better baseline than UVM. Graph preprocessing techniques, such as reordering, are useful even with compression, and we revisit them in Chapter 8. In the following experiment, we use the *zero-copy* approach from EMOGI [87].

In Fig. 7.1, we plot the performance of breath first search (BFS) in billions of traversed edges per second (GTEPS) on a Titan Xp GPU with 12 GiB memory. The graphs are

represented in the compressed sparse row (CSR) format and arranged in increasing order of their sizes. We use NVIDIA's high performance graph analytics framework, cugraph [90], for this experiment with light modifications for *zero-copy* memory transfers from the host for graphs that exceed the GPU's memory capacity. We see that the performance drops sharply when the graphs exceed the GPU's memory capacity. This is attributed to the fact that the GPU's internal memory bandwidth is $\sim$ 35x higher than the host-to-device interconnect's bandwidth (Table 7.1). We have marked three regions in Fig. 7.1:

- In the first region, CSR encoded graphs fit in memory. Here, cugraph's performance is quite good, and we do not need compression unless additional space (e.g., working data or outputs) is needed for the analytics kernel.

- The second region corresponds to graphs that exceed memory capacity. This is the region that stands to benefit the most from graph compression. Graphs in this region would fit in memory after compression and take advantage of the higher internal memory bandwidth.

- The last region is for massive graphs that would not fit in memory even after compression. However, due to the reduction in data transfers over the slow interconnect, compression is still useful in this region.

We note that while the overall trends in Fig. 7.1 are similar to those in Fig. 2.1 earlier, there are a few differences in the metrics. We use cugraph here instead of GraphBig since it has better performance. Second, in the UVM setting, we used a 64-bit representation for all the graphs. Since our goal is compression here, we use the most optimal CSR representation as a baseline, which translates to a mix of 32-bit and 64-bit types. All the graphs except the last two graphs can be represented with 32-bit unsigned types, and the last two graph use 64-bit types only for the row offsets; the column indices still be represented in 32-bits. With 32-bit types, we get a theoretical peak GTEPS rate of 3.03 for the out-of-memory region as the interconnect has a peak bandwidth of 12.11 GB/s. The primary

motivation for graph compression is to overcome this barrier imposed by the interconnect.

7.2 Background

7.2.1 GPU Architecture

The NVIDIA GPU architecture [91] consists of a number of cores known as streaming multiprocessors (SMs). All the cores have access to a common pool of global memory (12 GiB for the GPU used here), and each SM has additional local fast memory in the form of transparent and user managed caches known as *shared memory*. Each SM has a number of SIMD lanes, and a collection of threads, known as a *warp*, execute in lock-step fashion. Each SM has a large register file, and the scheduler switches warps on long latency operations such as loads from global memory. From the programmer's perspective, the global *grid* of threads is logically divided into *thread blocks*, where threads within a block have access to shared memory and synchronisation primitives. The GPU connects to the host via an interconnect such as PCI-e. Since a warp executes the same instruction in lock-step fashion, control flow *divergence* between threads can cause serialisation and reduce performance. The number of blocks resident on an SM, known as its *occupancy*, is determined by factors such as register pressure and shared memory usage. If the memory accesses by the threads in a warp are *uncoalesced*, the performance deteriorates. Achieving high performance on a GPU, particularly for irregular applications, requires careful consideration of the architecture's characteristics.

7.2.2 Graph Analytics on a GPU

A number of graph analytics applications use the common idiom of frontier expansion, filtering, and compaction. In Alg. 4, we show the basic structure of expanding a frontier in breadth first search (BFS). Given a frontier of active vertices, the GPU threads explore their neighbours in parallel and check if they have been visited before. Unvisited vertices are added to the next frontier. The complexity of the GPU implementation is abstracted in

the first two lines which are responsible for visiting the edges in parallel. Since the degree distribution of nodes in real-world graphs can be skewed [65], the primary challenge is mapping this expansion in a load balanced way onto the GPU's threads, which we cover in more detail in Sec. 8.2. The pattern in Alg. 4 also extends to applications such as single source shortest path (SSSP), PageRank (PR), betweenness centrality, connected components, and others. In SSSP, instead of a boolean visited flag, the condition tries to relax the distance of the destination vertex. In PR, there is no condition since each node distributes a fraction of its page-rank to all its neighbours in each iteration. In the context of graph compression, traversing an edge involves decompressing the edge. Once we have the destination of an edge, the rest of the algorithm is largely similar to the uncompressed case. These applications can also be cast as iterative Sparse Matrix Vector Multiply (SpMV) operations over a suitably defined semi-ring [6]. The vector is sparse in BFS as a node is active in the frontier only once during the traversal. In SSSP, the vector is denser since a node's distance value can be relaxed multiple times during the traversal and the same node would appear in many frontiers. The vector is fully dense in PageRank as every vertex is active in each iteration.

Algorithm 4 BFS_Level_Advance

```
1:  for all u ∈ current_frontier in parallel do
2:    for all (u, v) ∈ E in parallel do
3:      if (visited[v] == false) then
4:        old = atomic_or(&visited[v], true)
5:        if (old == false) then
6:          add_to_next_frontier(v)
7:        end if
8:      end if
9:    end for
10: end for
```

We use a collection of large graphs (Table 7.2) spanning web crawls [42, 43, 92, 93], social networks [45], a biological hypothesis network [46], and synthetic graphs [94]. Directed graphs are denoted with (d) whereas undirected ones are noted with (u). We also use

Table 7.2: Graphs used in this work

Graph	Category	Vertices	Edges
soc-livejournal (d)	Social Net	4.85 M	68.99 M
orkut (u)	Social Net	3.07 M	234.37 M
twitter (d)	Social Net	41.65 M	1.47 B
friendster (u)	Social Net	65.61 M	3.61 B
it-2004 (d)	Web	41.2 M	1.15 B
web-cc12-firstlevel (d)	Web	80.76 M	1.77 B
gsh-2015-h (d)	Web	68.66 M	1.80 B
sk-2005 (d)	Web	65.61 M	1.95 B
web-cc-12-host (d)	Web	89.11 M	2.03 B
uk-2007 (d)	Web	105.22 M	3.74 B
moliere-2016 (u)	Biomedical	30.22 M	6.68 B
kron_27 (d)	Kronecker	63.07 M	2.12 B
unrand_26 (d)	Erdős-Rényi	67.1 M	1.07 B

the symmetrised versions of directed graphs in the experiments, which are marked with a *_sym* suffix.

7.2.3 Scans and Searches

We make extensive use of two operations in this work: i) parallel scans, and ii) binary searches. For a list of n elements [a_0, a_1,..., a_{n-1}], an identity element I, and a binary associative operator $\oplus$, the exclusive scan returns the array [I, $(I \oplus a_0)$, $(I \oplus a_0 \oplus a_1)$,..., $(I \oplus a_0 \oplus a_1 ... \oplus a_{n-1})$]. This can be computed efficiently in parallel [95] with $\mathcal{O}(n)$ work and $\mathcal{O}(\log(n))$ depth. When the operator is summation, this is also known as a prefix sum. A variation of the regular scan is a segmented scan. In a segmented scan, there is an additional boolean flag array, F, where a 1 in the flag array denotes the start of a segment. The goal is to compute the scans within segment boundaries. This requires a modification to the binary operator [95], but is computationally similar to the regular scan. The binary searches we use in this work are parallel *bounded* searches, where each search finds the index of the largest value less than or equal to the search value. When combined, scans and searches serve an important role in load-balancing, which we cover in more detail in

Sec. 8.2.

7.3 Elias-Fano Encoding

Our graph compression format is based on Elias-Fano [96, 97] (EF) encoding for monotone sequences which dates back to 1970s. More recently, Vigna [98] described this encoding for compressing inverted indices, and he calls them *quasi-succinct* indices. Succinctness here refers to the classification of data structures that take close to the information theoretic lower bound in storage while still supporting efficient queries. We describe the main aspects of the encoding here and refer the reader to Vigna [98] for more details. Consider a monotonically increasing sequence of $n > 0$ natural numbers $0 \leq x_0 \leq x_1 ... \leq x_{n-1} \leq u$ where $u > 0$ is any upper bound on the last value. In standard binary encoding, such a sequence takes $n \lceil \log(u+1) \rceil$ bits of storage. The EF representation encodes it as follows:

- The lower $l = max\{0, \lfloor \log u/n \rfloor\}$ bits of each x_i are stored contiguously in the *lower-bits* array.
- The remaining upper bits of each x_i are stored in the *upper-bits* array as unary coded gaps with 1 as the stop bit.

See Fig. 7.2 for an example. The lower-bits array stores the 2 lower bits from each element. The remaining upper-bits are treated as integers (e.g., $1000_b \rightarrow 8$), and successive differences between them are encoded in unary (e.g., 8 - 5 = 3 which is 000) with a stop bit (1) and concatenated to form the final upper-bits array. The interesting property of this representation is that it takes at most $n(2 + \lceil \log u/n \rceil)$ bits in total to represent the sequence, which is quite close to the information theoretic lower bound for monotone sequences. The number of possible monotone sequences of size n from a universe of size u is $\binom{u+n-1}{n}$. Thus, the information theoretical lower bound is:

$$\left\lceil \log \binom{u+n-1}{n} \right\rceil \approx n \log \left(\frac{u+n}{n} \right) \tag{7.1}$$

We can see that the storage requirements are close to the optimal bounds, which leads to the "quasi-succinct" description in [98]. In Fig. 7.2, the original binary representation needs $6 * 8 = 48$ bits, whereas the EF representation needs 32 bits (16 each for the lower and upper bits arrays).

An alternate interpretation [99] of the upper-bits array is to view it as a histogram of keys. Since the sequence is monotonically increasing, one can recover the original values from this histogram. In this example, after separating the 2 lower bits, we are left with 4 bits for each value. There are $2^4 = 16$ possible values, or keys, for the upper half. However, we know that the last value is $1000 = 8$. Hence, we do not really need a histogram with 16 keys, but one with only 9 keys. To denote the separation between keys, we use a 0, and to store the frequency, we use the unary representation of the frequency with 1s. Since there are 9 keys, we need 8 separators (i.e., 8 zeros). The first key is 0, whose frequency is 2. Hence, in unary this translates to 11. This is followed by the separator 0. The next key is 1, whose frequency is 1 (which is still 1 in unary). This is followed by the separator 0, and so on. Thus, we end up with the same upper-bits array.

7.3.1 Decoding

To decode x_i, we need to recover and combine the lower and upper bits. The lower part can be readily recovered with a random access in the *lower-bits* array. The upper part of x_i can be recovered with a $select_1(i) - i$ operation on the *upper-bits* array, where $select_1(i)$ is defined as the operation that returns the position of the i^{th} (0-indexed) set bit from the start position. For example, to recover the upper bits of x_4, we compute $select_1(4) - 4 = 7 - 4 = 3$, since the position of the 4^{th} (0-indexed) set bit is 7. In the histogram analogy, this is finding the bucket in which x_4 belongs, which is bucket 3 (we pass 3 zeros which are bucket dividers). The $select$ operation is a foundational piece for information-retrieval systems and has been studied extensively in both theoretical and practical settings. Clark and Munro [100] provide a representation for succinct bitstreams (i.e., at most $o(n)$ in

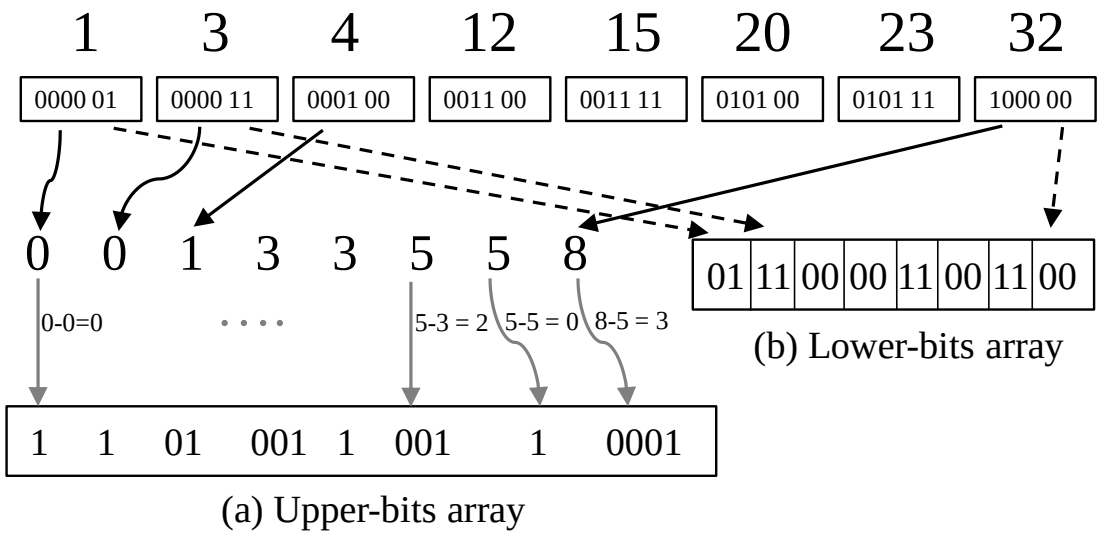


Figure 7.2: A monotone sequence {1,3,...,32} coded with Elias-Fano encoding. For $n = 8$ and an upper bound $u = 32$, we need $\lfloor \log 32/8 \rfloor = 2$ lower bits. Successive gaps between the upper bits are encoded in unary with 1 as the stop-bit to form the upper-bits array. The lower bits are concatenated.

extra storage) for constant time $select$ operations. In practice though, solutions that forego the property of succinctness have proven easier to implement and perform better. Both Elias [96] and Vigna [98] describe *forward pointers* as a straightforward way to get on-average constant time $select$ operations. Forward pointers store precomputed values of $select_1$ at multiples of some chosen quantum size. Now, performing $select_1(i)$ reduces to a $select$ starting at the nearest forward pointer boundary. We discuss forward pointers in more detail in Sec. 8.2.3. Since the density of 1s and 0s can vary throughout the bitstream, we cannot theoretically bound the number of reads needed starting at the boundary position, but it performs well in practice. Note that storing forward pointers has an overhead of $O(n)$ as opposed to $o(n)$ in succinct representations. Vigna presents efficient methods [101] for performing rank and select queries on bitstreams using broadword programming, which is also alternatively termed as SWAR (SIMD within a register). Recent Intel processors that support the Bit Manipulation Instruction set (BMI) make $select$ operations on 64-bit values relatively inexpensive, which is a technique that the folly [102] library leverages in their EF

implementation. Both these solutions focus on performing a single $select$ operation fast. That is, an application can replace its memory accesses with a call to the decoder, and the rest of the application would remain unchanged. Our problem is different in two important ways:

1. **Multiple Values**: In the GPU setting, we do not decode a single value in a list. Rather, a set of threads needs to decode a set of values. If each thread were to call $select$ independently, that would be wasteful since $select_1(i+1)$ would not use the work done during $select_1(i)$.

2. **Multiple Lists**: The set of values are not from the same list. In a graph application, we have a set of nodes (e.g., a frontier in BFS) whose neighbours we wish to explore. Each node's neighbour list is compressed with EF. The goal is to map this expansion of multiple non-contiguous compressed lists of different sizes in a load balanced way on the GPU's grid.

CHAPTER 8

ELIAS FANO GRAPH REPRESENTATION

8.1 Elias-Fano Graph (EFG) Format

We propose the Elias-Fano Graph (EFG) format as a GPU friendly compressed graph representation based on the EF encoding scheme. Each neighbour list is individually encoded with EF, and the overall graph is laid out in a format that is similar to CSR. The only requirement for encoding the neighbour lists with EF is that the original sequences should be in sorted order. While this is not strictly required in CSR, the neighbour lists are typically stored in sorted order since it facilitates other operations such as list intersection and hence can be converted to EFG directly.

In Fig. 8.1(c), we show the sample graph encoded in the EFG format. The representation consists of four arrays: `vlist`, `num_lower_bits`, `offsets`, and `data`. The first three arrays are indexed with the vertex id. The `vlist` array is similar to the one used in CSR, and it stores the exclusive prefix sum of the degrees of nodes. This array gives us constant time access to the degree of a node (i.e., $deg_i = vlist[i+1] - vlist[i]$), which is the number of elements in the compressed neighbour list. Unlike CSR, this array is not used directly for indexing the data. Instead, a separate `offsets` array stores the exclusive prefix sum of raw offsets into the compressed `data` array. The `num_lower_bits` array stores the number of lower bits used while encoding the vertex's neighbour list with EF. We use the folly [102] library for encoding the lists. The data is stored as `forward pointers`, `lower bits`, `upper bits`, in that order, in our implementation, and each section is byte aligned. There are no forward pointers in this example. To decode the neighbours of a node, we first recover its degree, the number of lower bits and offsets in the `data` array. For example, node 4 in Fig. 8.1 has a degree of 3, uses 1 lower bit per value,

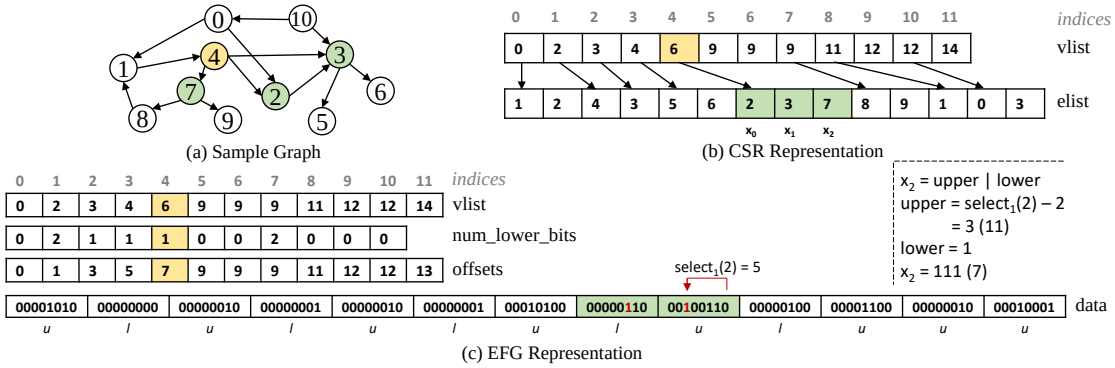


Figure 8.1: (a) A sample graph, (b) its CSR representation, and (c) its EFG representation. Node 4 and its neighbours are highlighted with yellow and green respectively. This is a small example for illustration which does not benefit from compression.

and its raw data is stored in 2 bytes. To recover x_2 (i.e., the third element), we compute $select_1(2) - 2 = 5 - 2 = 3 \equiv 11$ for the upper half[1] and 1 for the lower half for a total of $111_b = 7$, which is indeed the third neighbour of node 4.

The EFG format compresses the structure of the graph. A graph may have additional data associated with its nodes and edges. For instance, edge weights are stored in an additional array of size $|E|$ in CSR graphs. Compressing arbitrary floating point data is beyond the scope of this work, and we do not compress such auxiliary data.

8.2 Compressed Graph Traversal

We describe our implementation for GPU based traversal for EFG graphs in stages. The main goal here is to create a load-balanced implementation despite imbalances in the degree distribution. The problem of degree imbalance is shared with CSR graphs as well, and our top level decomposition is similar to a typical CSR based implementation. However, unlike CSR, ensuring a balanced mapping of threads to edges is only a part of the solution in our case since we also need to decompress the edges. In the CSR case, once a mapping of threads to edges is established, each thread can retrieve the edge in constant

[1] In Fig. 8.1, since we show the actual layout of bits in memory, the LSB is at the right end. Hence, select goes from right to left. Elsewhere in the dissertation, we use the prior convention of scanning bits from left to right for readability.

time from memory. In the EFG case, each thread block may get an equal number of edges, but these edges may come from different compressed lists. Hence, the crucial primitive for EFG is the decompression of multiple lists within a thread block. The fully general case of decompressing multiple lists is better understood through the building blocks that describe the decompression of a single list and partial lists within a block. In the following sections, we first describe the top-level load-balanced partitioning that is similar to the CSR case. This is followed by our decompression algorithm which is described as a progressive generalisation from a single list, to partial lists, finally to multiple lists. Finally, we discuss optimisations and extensions to other applications such as SSSP and pagerank.

8.2.1 Load Balanced Partitioning

Algorithm 5 Traverse

```
1: for all u ∈ current_frontier in parallel do
2:     for all (u, v) ∈ E in parallel do
3:         ...
4:     end for
5: end for
```

The main operation of interest in a GPU based traversal is the load-balanced expansion of the frontier of vertices, which is highlighted in Alg. 5. Consider the example in Fig. 8.2. The frontier consists of four arbitrary vertices. Their out-degrees are $\{2, 3, 2, 1\}$ for a total of 8 edges that emanate from this frontier. We need to visit these 8 edges to recover the destination vertices and eventually check a property such as the visited flag or a distance value. In an ideal mapping, 8 threads would explore the 8 edges. To achieve such a mapping, the following steps are taken: First, we compute the exclusive prefix sum (Sec. 7.2.3) of the degrees of these vertices. Next, each thread does a binary search in the exclusive sum array to find the index of the largest value less than or equal to the thread id. The returned index is an index in the frontier array, which is the vertex whose edge this thread needs to visit. For example, thread t_4 searches for 4 in the exclusive sum array, which returns the index 1

since 2 is the largest value ≤ 4. Hence, thread t_4 needs to visit a neighbour of v_1. Finally, to know which edge in particular to visit, the thread subtracts the exclusive sum from the thread id. Thread t_4 needs to visit 4 - 2 = 2 (i.e., 3rd edge) of node v_1. This approach was proposed by Bisson et al. [103]. If each thread were to do a binary search on the exclusive sum array in global memory, it would be inefficient. In practice, a strategy such as GPU Merge Path [104] can be used for the general problem of searching for multiple needles in a haystack in parallel on a GPU. Our own implementation uses the thrust [105] library's vectorised search functions for this phase. Irrespective of the implementation, notice that once a thread knows which edge it needs to visit, it can readily recover it in constant time in the CSR representation. Recall that the destination of the n^{th} edge of the i^{th} vertex in CSR is at `elist[vlist[i] + n]`. This does not hold true in the EFG representation. In our implementation for EFG graphs, we use this partitioning at the top level so that each thread block is responsible for roughly equal number of edges. However, within a thread block, we need a new implementation for decompressing the data. In this example, if we want to split the work between two thread blocks of 4 threads each, the partitioning can tell us that thread block 0 needs to visit all of v_0's edges and the first two edges of v_1, whereas thread block 1 needs to visit the last edge of v_1 and all of v_2 and v_3's edges. This would be followed by a thread block level implementation of decompression.

We have been using an edge as the unit of work. That is, our ideal case has been to split the edges equally between threads. This works well in the uncompressed case since traversing an edge is always one read from memory. With compression, this definition does not apply strictly, and even with an equal split of edges, we would not split the work equally. This is because the actual work required to decode an edge is not uniform. One thread block may go through more (compressed) data than another before it decodes the same number of edges. Even within a block, threads may do different amounts of work. Alternatively, one can view data as the unit of work. However, that just shifts the imbalance to the uncompressed domain. If we split the (compressed) data equally between thread

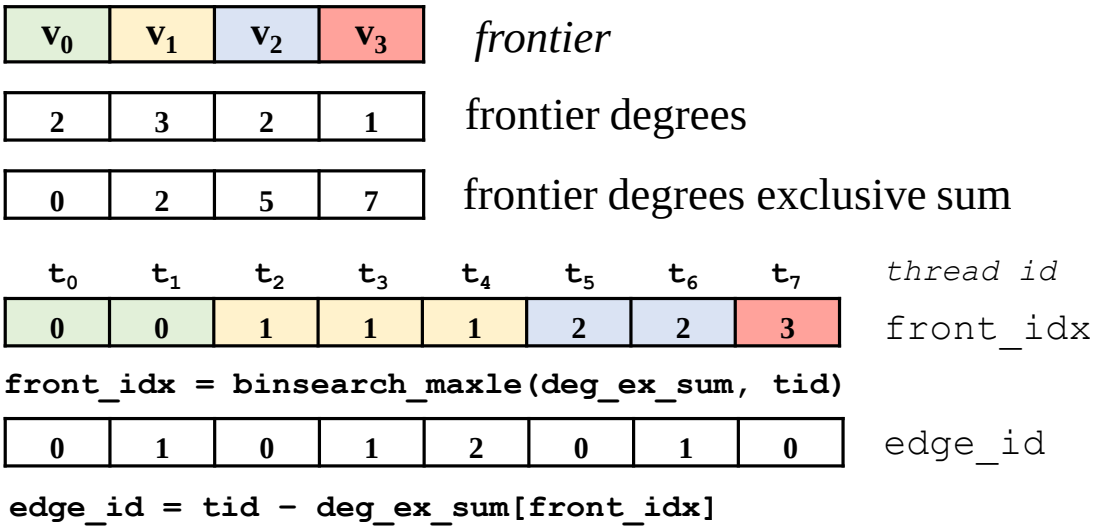


Figure 8.2: Mapping of edges to threads in frontier expansion. `binsearch_maxle` returns the index of the largest value less than or equal to the search value.

blocks, the thread blocks would produce unequal number of edges, and hence do unequal work in the analytics application that uses the edges. We choose the former approach of splitting edges equally since it is closer to the uncompressed case from the application's viewpoint, and it has one additional advantage: It allows us to use forward pointers in EF encoding, which are also used in the serial CPU based implementation. That is, our GPU implementation does not require any modifications to the data structure. We discuss the use of forward pointers in more detail in Sec. 8.2.3.

In summary, the top level decomposition partitions the edges equally between thread blocks. The remainder of the problem can be formulated as that of decompressing multiple neighbour lists, where the first and the last list may be partial, within a thread block. We describe the solution as a progressive generalisation of specific cases.

8.2.2 Decompressing A Single List

Consider the specific case of decompressing a single list within a thread block. This is, in fact, sufficient for implementing the traversal, albeit in an inefficient way due to skews in the degree distribution. Instead of a load-balanced split of edges, as described earlier, we

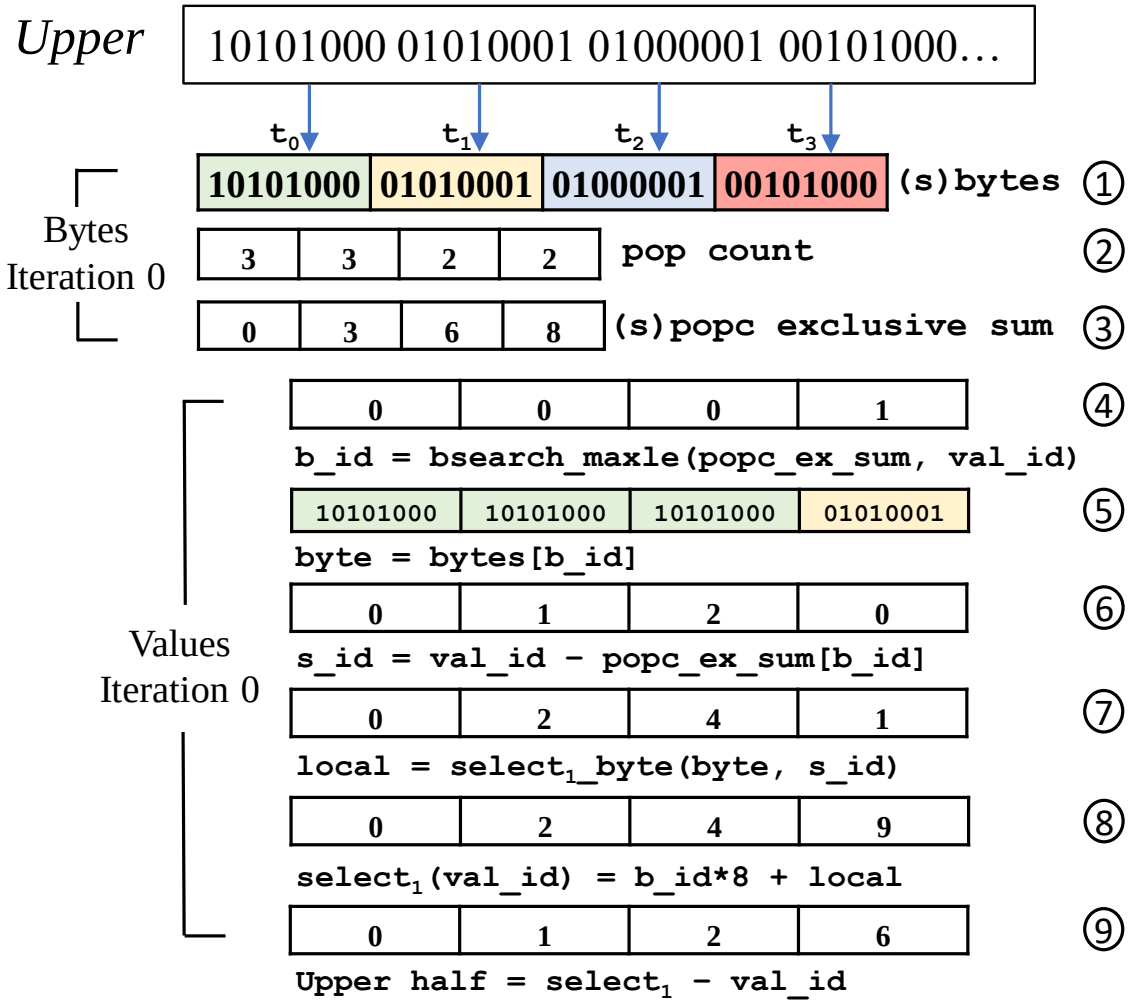


Figure 8.3: Decompressing a single list within a thread block. *Upper* is the upper bits array. The threads collectively compute $select_1(i) - i$. Shared data structure are marked with (s). `val_id` is the same as `thread_id` in the first iteration.

can assign a thread block to each node in the frontier, and the thread block decompresses that node's edge list. Recall from Sec. 7.3.1 that we are mainly interested in computing $select_1(i) - i$ for each of the set bits in the upper-bits array, where $select_1(i)$ returns the position of the i^{th} set bit in a bitstream. Since the lower-bits array can be accessed in constant time, we focus only on the upper-bits array. The pseudo-code for decompressing a single list is show in Alg. 6, and we show a corresponding example in Fig. 8.3.

In this example, a thread block of 4 threads collectively decompresses a list by computing $select_1(i) - i$ for each set bit. The number of threads in a block is denoted by `DIMX`

Algorithm 6 decompress_single_list (upper, n_bytes)

```
 1: __shared__ s_popc_exsum[DIMX];
 2: __shared__ s_bytes[DIMX];
 3: prev_vals = 0;
 4: b_iters = ceil(n_bytes / DIMX);
 5: for (i=0; i ¡ b_iters; i++) do
 6:    byte_id = i * DIMX + thread_id;
 7:    byte = (byte_id ¡ n_bytes) ? upper[byte_id] : 0;
 8:    s_bytes[tid] = byte;                                                  ①
 9:    popc = __popc(byte);                                                  ②
10:    total_vals = do_ex_sum(popc, s_popc_exsum);                           ③
11:    CTA_sync();
12:    val_iters = ceil(total_vals / DIMX);
13:    for (j = 0 ; j ¡ val_iters; j++) do
14:       val_id = j * DIMX + thread_id;
15:       if (val_id ¡ total_vals) then
16:          tb_id = bsearch_max_le(s_popc_exsum, val_id);                   ④
17:          target = s_bytes[tb_id];                                        ⑤
18:          s_id = val_id - s_popc_exsum[tb_id];                            ⑥
19:          select_result = select_1_byte(target, s_id);                    ⑦
20:          bits_before_me = (i*DIMX + t_byte_id)*8;
21:          select_result += bits_before_me;                                ⑧
22:          global_val_id = prev_vals + val_id;
23:          upper_half = select_result - global_val_id;                     ⑨
24:          lower_half = get_lower_half (global_val_id);
25:          decoded_val = combine(upper_half, lower_half);
26:          // Use value in analytics
27:       end if
28:    end for
29:    prev_vals += total_vals;
30:    CTA_sync();
31: end for
```

in the code. The upper bits array (*Upper*) resides in global memory. Each thread begins by ① loading a byte to a shared bytes array. Shared data structures reside in a fast user managed cache and are accessible by all threads within the block. Next, each thread ② computes a population count of the set bits in its respective local byte with the `popcount` instruction. The counts are {3, 3, 2, 2} here. The number of set bits in a byte is also the number of values that the byte will eventually produce. Since these values are not uniform, we need a load balanced split similar to the approach described earlier for splitting edges based on unequal degrees. Hence, the threads collectively compute the ③ exclusive prefix sum (Sec. 7.2.3) over their respective pop-counts. The results are stored in a shared array. The exclusive scan also returns the total pop-count (10 in this example), which is the total number of values to be decoded. This starts the inner loop where each thread decodes one value in each iteration. We show the first iteration in the example.

In the first iteration, the index of the value to be decoded is the same as the thread id (line 14, Alg. 6). Thread *i* needs to find the position of the i^{th} set bit. Since we have the exclusive prefix sum of pop-counts, each thread does a ④ binary search in the array to narrow the search to a ⑤ target byte. In the example, we see that the first three threads get the first byte whereas the fourth thread gets the second byte. The difference between the search parameter and the exclusive sum gives the new search parameter for ⑥ selecting a bit within a byte, called $select_1\_byte$ hereafter (e.g., thread t_2 finds the position of the 2nd set bit in `10101000`). A byte has 256 possible values, and for each value, there can be at most 8 positions. Hence, $select_1\_byte(i)$ is implemented with a ⑦ lookup table of 2 KiB in constant memory. The number of preceding bits in the list is ⑧ added to the result of $select_1\_byte$ to produce the global $select_1$ value, and ⑨ subtracting the index of the value from it gives the upper-half of the decompressed value. This is combined with the lower half (not shown) and used in the application.

Notice that we need the binary search in the inner loop because we want the 1:1 mapping of threads to values to continue through the decompression into the application itself.

If we forego this property, each thread could instead decode the values within its local byte. This still needs the prefix sum, but avoids the binary search. We use the former approach as it fared better in our experiments. Since the searches are in shared memory, the cost of searching is not too high.

8.2.3 Decompressing A Partial List

Assigning a list to a block has obvious limitations since some lists can be much longer than others, particularly in real-world graphs with a power-law degree distribution. The next logical step is to split long lists across thread blocks in order to avoid over-subscription of thread blocks. Since a list can span multiple blocks, the problem can be formulated as that of decoding values in some range $[a, b]$ within a list, where $0 \leq a \leq b < n$ for a list with n elements. This is achieved with the use of forward pointers. The conventions herein are based on the folly [102] library.

For a list of size n and a quantum parameter $k > 0$, we store $\lfloor n/k \rfloor$ forward pointers that enable fast $select_1(i)$ operations for $i = \{k-1, 2k-1, ..., \lfloor n/k \rfloor k-1\}$. The parameter k is 1-indexed since $k = 0$ implies that there are no forward pointers. For instance, $k = 8$ stores values for $select_1(8 - 1 = 7)$, $select_1(15)$, and so on. The pointers actually store $select_1(i) - i$ rather than $select_1(i)$ since it takes fewer bits, and i can be added later if needed. Consider the example in Fig. 8.4, where forward pointers are stored for $k = 8$, and a thread block of 4 threads needs to decode values $[x_{12}, x_{20})$ in the list. The closest preceding pointer for x_{12} is at $\texttt{forward}[\lfloor (12 + 1)/8 \rfloor - 1]$, which is the first pointer, and it corresponds to x_7. We get a bit position of $4 + 7 = 11$, as shown in the figure. Similarly, the last pointer of interest corresponds to x_{23}, which gives a position of 53. Thus, this thread block only needs to scan between bits 11 and 53. Since we work with byte alignment, we need a mask when we load the first byte to mask out any preceding 1s that are set within the byte. The overall structure of the algorithm remains similar to Alg. 6 with a few differences. In the inner loop, the threads do not start with $select_1(0)$. Instead, they would start with

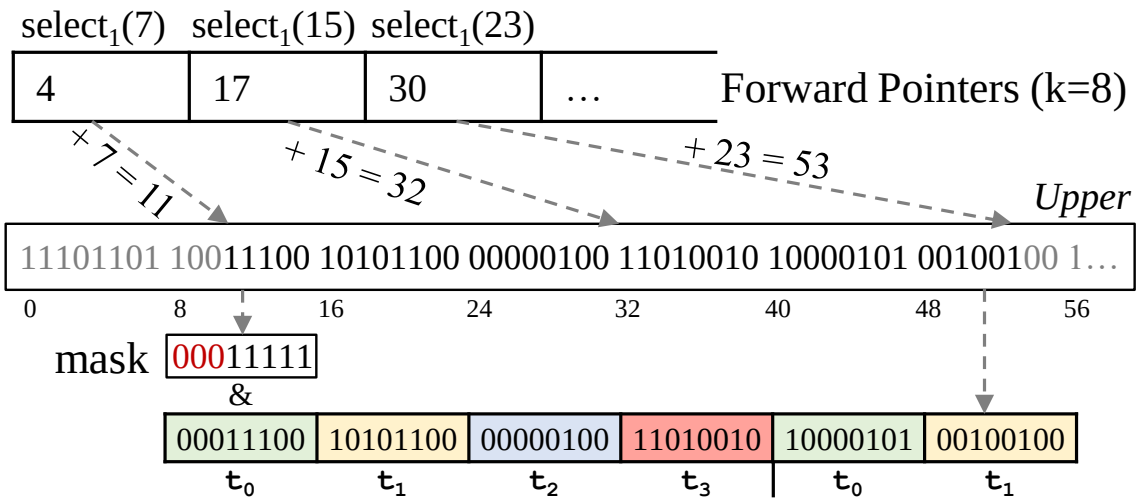


Figure 8.4: Decompressing a partial list within a thread block. Forward pointers store the value of $select_1(i) - i$ at regular intervals ($k = 8$ here). The thread block loads the bytes between two boundary pointers.

$select_1(12)$ in this case, but since the starting bit position is already at $select_1(7)$, the first thread starts with $select_1(12 - 7 = 5)$. The number of skipped bits is added to the result.

Forward pointers are crucial for performance. In the absence of forward pointers, each thread block that needs to start in the middle of a list would have to scan bytes from the beginning of the list. The outer loop would load bytes from the beginning, and threads would compute the prefix sum of popcounts, but if the target values were not in range, they would not do any useful work, and move on to the next set of bytes. Forward pointers are also used similarly in a serial CPU implementation since performing a single random $select_1$ involves skipping over the nearest forward pointer. Our implementation choice of splitting the work in terms of edges allows us to reuse the same convention in pointers. Splitting the work in terms of compressed data would require a slightly different dictionary of pointers, namely a dictionary that stores the total number of set bits up to a position at regular intervals.

Splitting large lists across blocks helps with over-subscription, but not with under-subscription since small lists are still assigned to individual blocks. The last missing piece in the general solution is the ability to decode multiple lists within a block, which we de-

scribe next.

8.2.4 Decompressing Multiple Lists

The load-balanced partitioning described in Sec. 8.2 assigns an equal number of edges to different thread blocks. Hence, each thread block is responsible for decoding a number of lists. The single-list solution in Alg. 6 changes in a few ways to account for multiple lists. First, we now need an additional outer loop for lists. That is, we need three levels of nesting that go from lists to bytes to decoded values, and the bytes come from multiple lists instead of a single list. Second, notice that at the innermost level, a thread's view is extremely local. Each thread needs to compute $select_1(i) - i$ for some i, and in order to go from $select_1\_byte$ on a local byte to the global decoded value, it needs two pieces of information: i) What the position of the byte in the list is, and ii) What the global index (i) of the value in the list is. The position of the byte in the list is needed because the local select value is added to the number of preceding bits (`bits_before_me` in Alg. 6, line 21). The global index, i, is also needed since it needs to be subtracted from the result of $select$. Since we have one set bit for each value in the upper-bits array, i is also the total number of set-bits up to the thread's local byte. This can be seen in line 22, where the `global_val_id` is computed by adding `prev_vals` to the local id, where `prev_vals` keeps a running total of number of set bits seen so far. In the single list case, both of these were relatively straightforward to compute. When we have multiple lists, each thread instead needs to know the number of preceding bits and the global index of the value *within* the list whose byte it is decoding. This requires a segmented prefix sum (Sec. 7.2.3), where the segments represent list boundaries. We look at an example next to explain this.

In Fig. 8.5, a thread block of 6 threads decompresses multiple lists. Steps marked with $\star$ are different from the single-list case. Like the single-list case, each thread needs to load a byte to the shared bytes array. However, since the bytes now come from multiple non-contiguous lists of different sizes, we use the familiar idiom of a prefix sum over

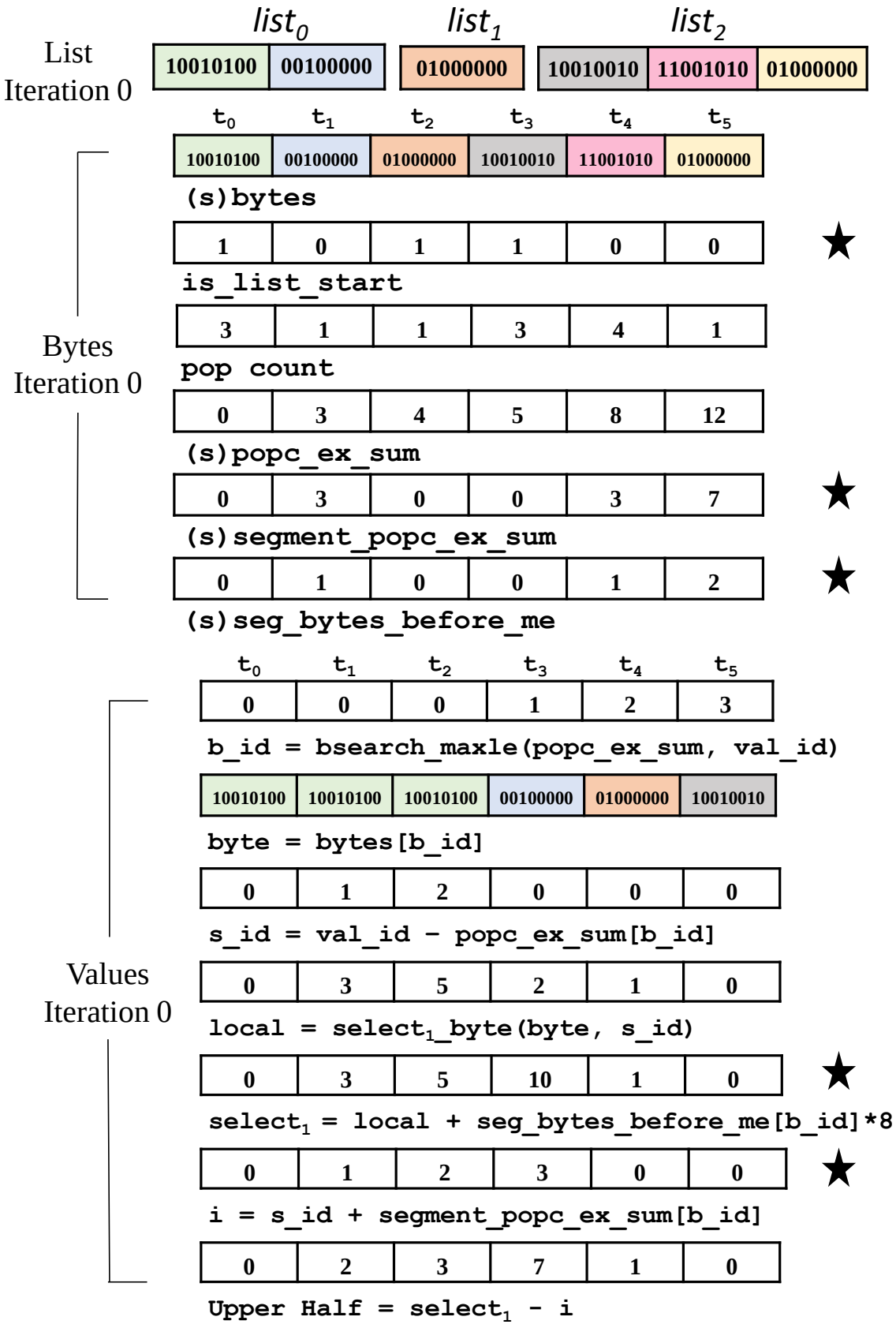


Figure 8.5: Decompressing multiple lists within a thread block. Steps marked with $\star$ are different from the single-list case.

the number of bytes per list followed by a binary search to map the bytes to threads and use the offsets array in the EFG graph to find the start positions of lists. This step is omitted in the figure, and the figure starts at the step where each thread has loaded its byte to the shared bytes array. Like the single-list case, these bytes would produce a number of values, and we need an exclusive prefix sum over the local popcounts for use in the innermost loop. In addition, we need to mark the list boundaries. Threads that start a new list mark a flag in the `is_list_start` array. For eg., thread t_2 starts $list_1$, and marks its bit as 1. This flag array is used in conjunction with the popcounts to create an additional segmented prefix sum array, where each prefix sum runs only within the list. While thread t_4's block-wide exclusive sum is 8, its exclusive sum within the list is 3. Similarly, the `seg_bytes_before_me` array keeps track of the number of preceding bytes *within* the list. In the innermost loop for decoding the values, the threads identify their target bytes by searching the block-wide exclusive array like the single-list case. After computing the local $select_1\_byte(i)$ value, each thread looks up the number of preceding bits within the list and adds it to get the global $select_1$ value. For eg., since t_3's target byte id is 1, which has one preceding byte in the list, 8 is added to its local select value of 2 for a total of 10. On the other hand, t_4's target byte is byte 2, which is the start of a list. Hence, nothing is added to its local select value. Similarly, the segmented exclusive sum of the target byte is added to the id within the byte to get the global id for the value. Finally, the upper half is computed as $select_1(i) - i$ and combined with the lower half to recover the value.

We looked at decompressing multiple complete lists within a block, but it can be extended naturally to deal with partial lists at the boundaries by using forward pointers (Sec. 8.2.3). Interested readers can find more details in our published code.

8.2.5 Additional Optimisations

In the general multi-list case, the lists that a thread block decompresses may be scattered across memory. Some lists may be smaller than a cacheline, or may spill into a cache-

line, which leads to read amplification. Similarly, the prefetcher is less effective if the read access pattern is random. In breadth first search, if we were to sort the nodes in a frontier, the partitioning (Sec. 8.2) of the frontier between thread blocks would impart a sorted order to the lists that are assigned to the thread blocks. Note that the lists can still be non-contiguous, but the thread blocks touch non-overlapping regions of memory. However, sorting the frontier at each level is expensive, and the cost outweighs the benefits of improved locality. Fortunately, we do not need an exact sort since this is just an optimisation and does not affect correctness. Radix-sort, specifically the least-significant-digit-first (LSD) variant, is a non-comparison sort that sorts the keys one bit at a time as it sweeps from the least to the most significant bit. We use the CUB [106] library's GPU implementation for radix-sort to sort only the higher order bits. We sort 65% of the bits (i.e., we pretend as though the lower 35% bits do not exist) while sorting the frontier. We see an average improvement of 9% (max 33%) in runtime from this optimisation.

8.2.6 SSSP and PageRank

The general traversal pattern extends to many applications, and we extend it to single source shortest paths (SSSP) and PageRank here. The standard approach for implementing a parallel SSSP on GPUs is to use the Bellman-Ford algorithm. It does asymptotically more work than Dijkstra's sequential algorithm, but lends itself well to parallel execution whereas Dijkstra's algorithm does not. More advanced algorithms such as delta-stepping SSSP [83, 84] have been proposed, but they are challenging to implement on GPUs and are outside the scope of this work. The SSSP implementation is similar to BFS except that instead of a boolean visited property, we need to relax a floating point distance value. Unlike BFS though, a node's distance can be relaxed multiple times in the same iteration by any number of its incoming edges. Hence, if relaxed nodes are added to the frontier directly by the parent (line 6, Alg. 4), the frontier would have duplicates. They can be pruned, but it requires enough memory to accommodate a frontier of size $|E|$, which is problematic for

large graphs. Hence, we use a different approach where we mark the relaxed nodes atomically in a bitmap of size $\mathcal{O}(|V|)$ and use a parallel scatter to create the frontier from the bitmap. As a side-effect, this approach also creates a sorted frontier. Note that the edge weights in the input graph also require $\mathcal{O}(|E|)$ in storage since we compress the graph structure but not the weights in EFG. Hence, SSSP gets in the out-of-core regime well before BFS. Compressing weights is outside the scope of this work. In PageRank, all the nodes are active in each iteration, and we do not need a frontier (i.e., the frontier comprises all the nodes). The page-rank value of a destination is updated atomically once the edge is decoded.

8.3 Related Work

The literature on graph compression methods is vast [5]. Compression can be seen as a pipeline where different approaches are used successively. For instance, hierarchical schemes that collapse a portion of the graph into a smaller structures, (e.g., virtual-nodes [107]), can be used independently from the work described in this paper. Perhaps the most widely-used method for compressing large web-graphs is BV [108]. BV exploits locality within lists and similarity across lists. The gaps between neighbours are coded with a variable length code and reference chains are used for capturing the similarity between different lists. The reason for using gaps is that even for large numbers, gaps between successive numbers can be small and use only a few bits. Reference chains and variable length codes are sequential in nature, and it is difficult to adapt the decompression for GPUs. Gap based encoding is widely used, and a number of prior works reorder the graph to reduce gaps. LLP [53] identifies clusters that are similar in graph structure and labels the nodes using label propagation. Shingle [49] and BP [50] reorder graphs by formalising the problem as variants of the minimum linear/log arrangement problems. A few recent works focus on the decompression performance in parallel CPU setting. Ligra+ [109] extends the CPU based parallel graph processing framework, Ligra [110], to compressed graphs. LogGraph [111]

uses ideas from succinct representations and demonstrates high performance on the parallel GAP [94] benchmark suite.

The only prior work, and the current state of the art, for GPU based traversals on compressed graphs is by Mo Sha et al. [10]. The compression scheme breaks lists into intervals (contiguous runs of values), and residuals, followed by gap transformation, and gaps are encoded with a variable length code. Their GPU implementation for traversals use a two-phase strategy with work-stealing. The authors call this encoding CGR, and we compare our work against it. We use the term CGR interchangeably to refer to the graph representation as well as the GPU BFS implementation in [10].

CHAPTER 9
GRAPH COMPRESSION RESULTS

9.1 Results

We evaluate CSR, EFG, and CGR graph representations in a number of experiments. We use cugraph [90] for CSR graphs and the reference implementation in [10] for CGR graphs. All experiments are performed on a Titan Xp GPU with 12 GiB memory. We choose 100 random starting nodes for BFS and SSSP and average the results. Edge weights are initialised to random floating point values between 0 and 1 for SSSP. PageRank runs are capped at 50 iterations. We modified cugraph lightly to support out-of-core processing in BFS and SSSP. CGR does not support out-of-core processing, and experiments where CGR is unable to process large graphs are marked as 'did not run' (DNR). The forward pointer quantum parameter k is set to $k = 512$ in EFG.

9.1.1 Compression Ratio

We show the compression ratio of EFG and CGR relative to CSR in Fig. 9.1, and absolute numbers are provided in Table 9.1 for reference. The CSR graphs use the smallest possible fixed width type (typically 4 bytes) for the graphs. The graphs are grouped by categories in Fig. 9.1 as social, web, and other graphs. EFG achieves a compression ratio of 1.55x over CSR, and the compression is quite consistent across graphs. CGR shows a significant skew in the compression ratio where it excels at web-graphs (e.g., `sk-2005_sym` by a factor of 7.1x), but in other graphs, its ratio is lower than EFG. CGR's average compression ratio across all graphs is 1.86x. The reason for the disparity in web-graphs is two fold: i) Web-graphs have strong locality where long runs of contiguous values are common. The interval and residual representation in CGR lends itself well to this structure. ii) EF encoding

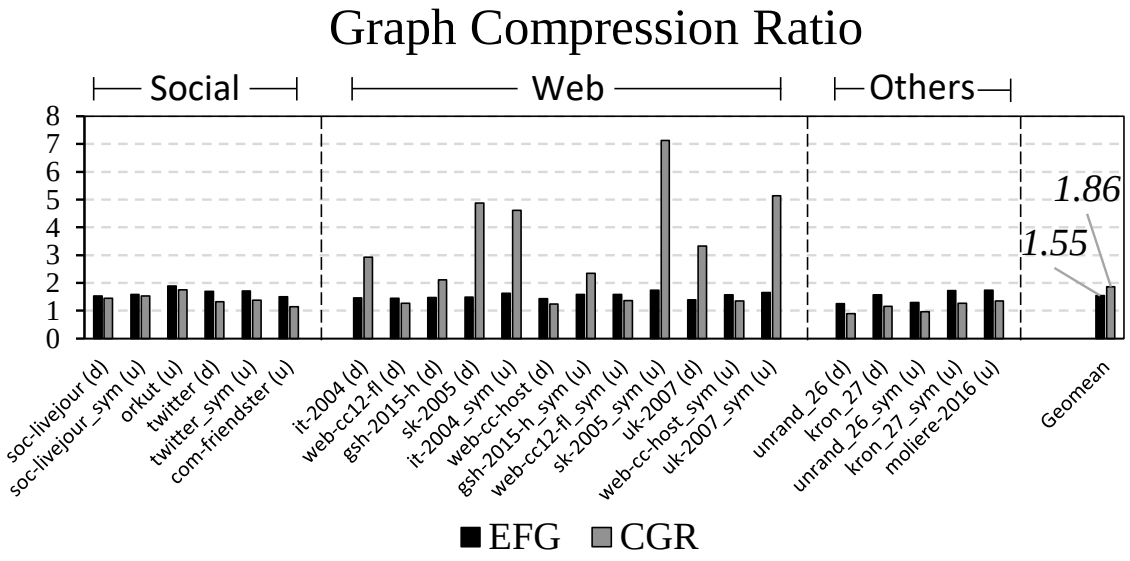


Figure 9.1: Compression ratio for EFG (this work) and CGR [10] relative to CSR. CGR excels at web-graphs whereas EFG is better in other categories.

has limitations in compressing such sequences, and solutions [112] that address it exist, although they were not incorporated here. We discuss them in Sec. 9.2. Notwithstanding the lower compression in web-graphs, EFG has other benefits. First, it is inexpensive to estimate EFG's storage requirements since the upper bound on storage in EF only depends on the number of elements and the largest value (Sec. 7.3) in each list. That is, we do not need to compress the graph to know how well it will compress with EFG. Second, the runtime performance for traversals is better for EFG in almost all cases, which we discuss next.

9.1.2 BFS Performance

We show the relative performance of BFS for the different representations in Fig. 9.2 and report the absolute runtime in Table 9.1. As noted in Sec. 7.1, the set of graphs break down into three categories. The first set of small graphs fit in memory even in the CSR representation, and here cugraph's [90] implementation performs the best. EFG achieves 0.76x of cugraph's performance, which is better by a factor of 2.1x than CGR. The next set of graphs exceeds the memory capacity in the CSR representation. We can see that EFG performs the

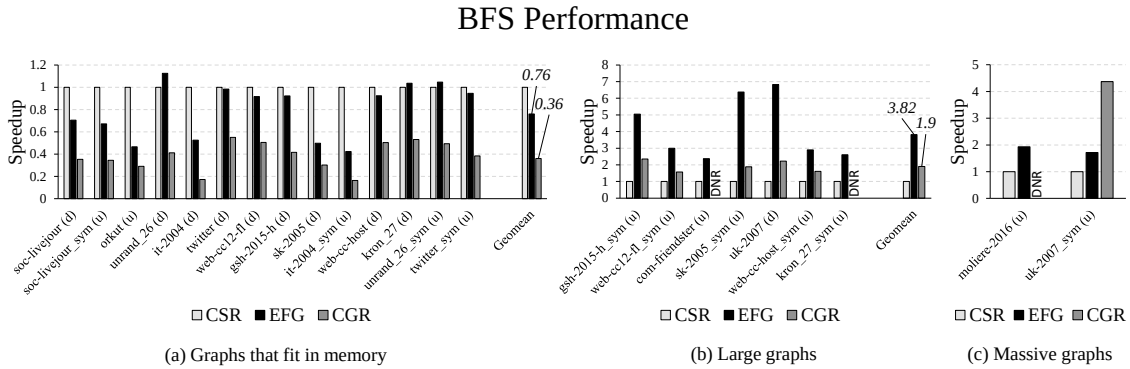


Figure 9.2: BFS performance relative to CSR (higher is better) on a Titan Xp GPU with 12 GiB memory

best here as all the graphs fit in memory after compression. EFG has an average speedup of 3.8x (max 6.8x) over the out-of-core CSR implementation and a 2x speedup over CGR in this region. Notice that EFG's runtime is lower that CGR even for web-graphs that compress really well with CGR. For example, CGR compresses `sk-2005_sym` from 13.75 GiB down to 1.93 GiB, whereas EFG compresses it down to 7.9 GiB. However, EFG's runtime is 3.4x lower than CGR (323 ms v/s 1098 ms). The last two graphs in the set take more than 12 GiB in storage even after compression with EFG, and we are in the out-of-core regime for both CSR and EFG. However, compression is still beneficial here since it reduces the volume of data transferred over the interconnect. EFG sees a speedup of 1.8x over CSR here. For the last two graphs, CGR does not run for `moliere-2016`, but it compresses the `uk-2007_sym` web graph down to 4.9 GiB, which fits in memory. While EFG performs better than CGR across all other graphs, the last graph is the sole exception since the CGR graph fits in memory whereas the EFG one does not. On average, EFG performs better by 2x compared to CGR.

9.1.3 SSSP and PageRank Performance

We show the performance of SSSP and PageRank as measured in billions of traversed edges per second (GTEPS) in Figs. 9.3 and 9.4, respectively. CGR is not evaluated for these kernels since the implementation was not available. Since SSSP requires an additional

Table 9.1: Graph size and BFS runtime on a Titan Xp GPU

Graph	$\|V\|$	$\|E\|$	Size in GiB (Runtime in ms)		
			CSR [90]	EFG	CGR [10]
soc-livejour (d)	4.85 M	68.99 M	0.28 (8)	0.18 (11)	0.19 (22)
soc-livejour_sym (u)	4.85 M	86.22 M	0.34 (10)	0.21 (14)	0.22 (28)
orkut (u)	3.07 M	234.37 M	0.88 (13)	0.47 (28)	0.5 (45)
unrand_26 (d)	67.1 M	1.07 B	4.25 (525)	3.4 (467)	4.72 (1277)
it-2004 (d)	41.2 M	1.15 B	4.44 (41)	3.05 (78)	1.52 (239)
twitter (d)	41.65 M	1.47 B	5.63 (234)	3.33 (238)	4.23 (425)
web-cc12-fl (d)	80.76 M	1.77 B	6.92 (249)	4.76 (272)	5.48 (493)
gsh-2015-h (d)	68.66 M	1.80 B	6.97 (160)	4.73 (174)	3.3 (385)
sk-2005 (d)	65.61 M	1.95 B	7.45 (57)	5.02 (115)	1.53 (190)
it-2004_sym (u)	41.2 M	2.06 B	7.87 (82)	4.85 (193)	1.7 (501)
web-cc-host (d)	89.11 M	2.03 B	7.93 (303)	5.52 (328)	6.36 (603)
kron_27 (d)	63.07 M	2.12 B	8.15 (511)	5.18 (494)	7.01 (962)
unrand_26_sym (u)	67.1 M	2.14 B	8.25 (793)	6.39 (758)	8.59 (1610)
twitter_sym (u)	41.65 M	2.40 B	9.11 (348)	5.34 (368)	6.61 (906)
gsh-2015-h_sym (u)	68.66 M	3.04 B	11.62 (1824)	7.33 (361)	4.94 (776)
web-cc12-fl_sym (u)	80.76 M	3.38 B	12.92 (2140)	8.17 (713)	9.48 (1360)
com-friendster (u)	65.61 M	3.61 B	13.7 (2387)	9.15 (1006)	11.98 (DNR)
sk-2005_sym (u)	65.61 M	3.63 B	13.75 (2062)	7.9 (323)	1.93 (1098)
uk-2007 (d)	105.22 M	3.74 B	14.32 (1444)	10.31 (212)	4.3 (648)
web-cc-host_sym (u)	89.11 M	3.87 B	14.76 (2441)	9.37 (842)	10.89 (1519)
kron_27_sym (u)	63.07 M	4.22 B	15.97 (2600)	9.23 (997)	12.61 (DNR)
moliere-2016 (u)	30.22 M	6.68 B	25.1 (4149)	14.5 (2148)	18.65 (DNR)
uk-2007_sym (u)	105.22 M	6.62 B	25.47 (4859)	15.43 (2825)	4.96 (1113)

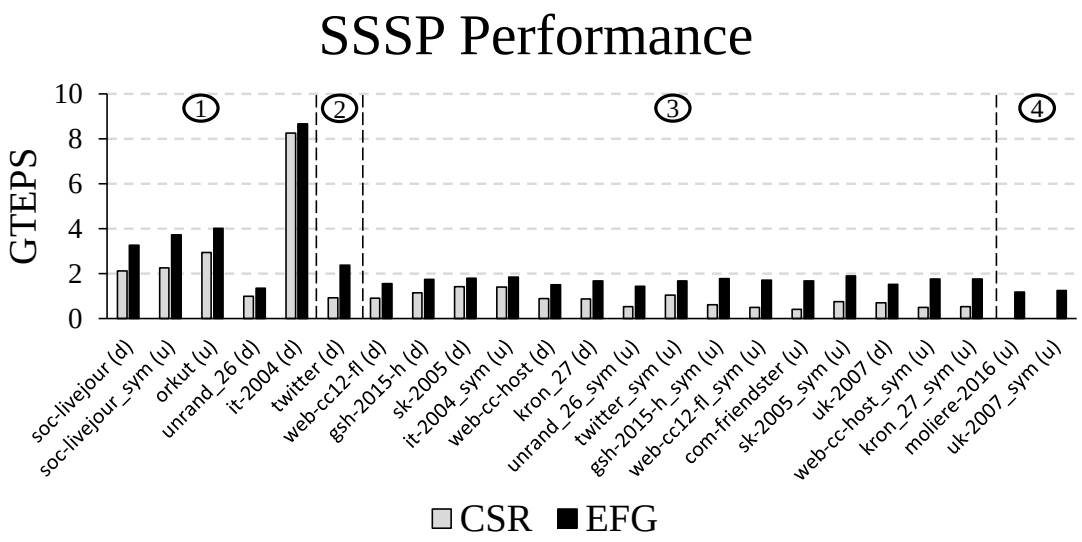


Figure 9.3: SSSP performance measured in GTEPS. Regions: (1) CSR and EFG graphs fit in memory. (2) EFG fits entirely but CSR does not. (3) EFG fits but edge weights do not. (4) EFG does not fit.

edge weights array, only small graphs fit entirely in memory, even with compression as weights are not compressed. The graph structure fits in memory for the majority of EFG graphs, but weights are streamed from the host. EFG shows a speedup of 1.38x over CSR for small graphs (region 1 in Fig. 9.3) and 2.25x for larger graphs (regions 2-3 in Fig. 9.3). While compression helps and performs better than the uncompressed case, we are still bottlenecked by the interconnect. PageRank's performance (Fig. 9.4) shows a similar trend as BFS in that cugraph's in-memory implementation for CSR performs better than EFG. PageRank is not evaluated for CSR for large graphs since an out-of-core version was not available.

9.1.4 Graph Reordering

Graph reordering is a common preprocessing technique that is used for different purposes. In the context of graph compression, reordering methods relabel the nodes to reduce gaps between successive neighbours since smaller gaps can be encoded more efficiently. Graph reordering can also be used to improve the locality and performance since graph applica-

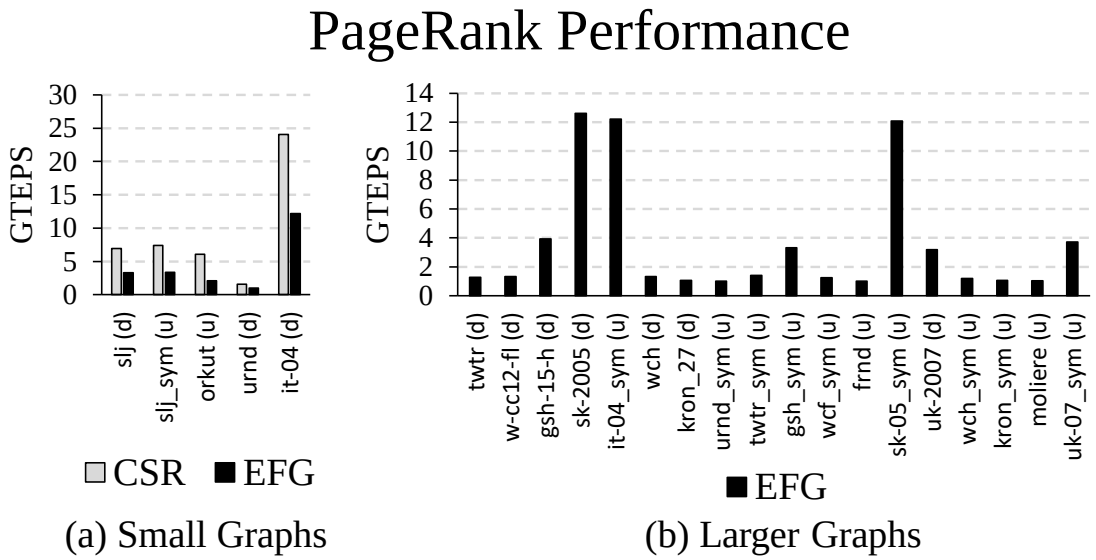
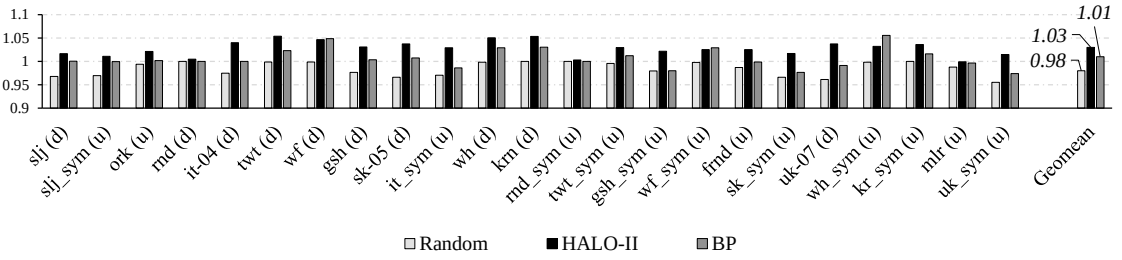


Figure 9.4: PageRank performance measured in GTEPS

tions tend to have irregular memory access patterns. We use two graph reordering methods - one that optimises for lower gaps and one that optimises for locality, and evaluate their impact on EFG and CGR graph representations. We use BP [50] as a compression friendly reordering method and HALO [88] as a locality friendly method. BP uses a recursive graph bisection strategy to optimise a cost function that reduces gaps. HALO uses an ordering based on harmonic centrality to improve the locality in traversal-like patterns so that nodes in a frontier are close to each other. We use the PISA [81] framework to implement BP and our own implementation for HALO. We also evaluate random ordering as a pathological case since random ordering destroys all locality. The impact of different orderings on compression ratio and BFS' runtime performance is shown in Figs. 9.5 and 9.6. The main observations are as follows:

Compression Ratio: EFG does not use a gap-based encoding. Hence, it is not affected by the distribution of gaps and its compression ratio is virtually unchanged for all the ordering methods. Note that random ordering does not negatively impact the compression ratio. The storage bounds for EF only depend on the largest value in a list and the number of elements. On the other hand, CGR benefits from lower gaps, and its compression ratio

(a) EFG Compression ratio relative to Natural Ordering



(b) CGR Compression ratio relative to Natural Ordering

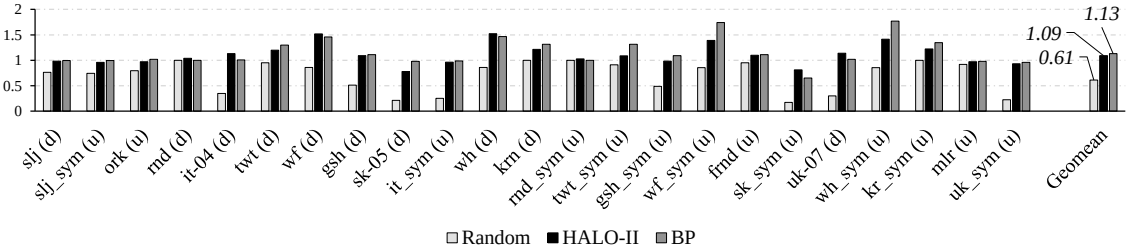
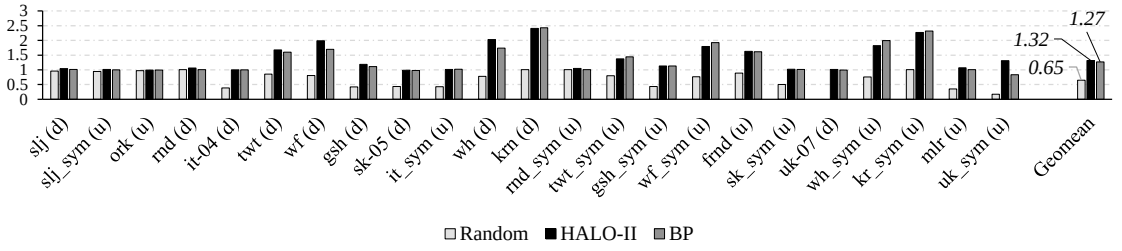


Figure 9.5: Impact of graph reordering on compression ratio

(a) EFG BFS Speedup relative to Natural Ordering



(b) CGR BFS Speedup relative to Natural Ordering

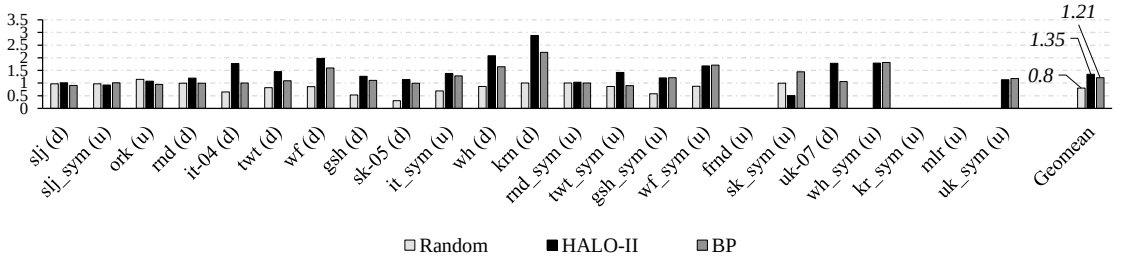


Figure 9.6: Impact of graph reordering on BFS performance

Table 9.2: Forward Quantum Sensitivity

Graph	Size (Runtime)		
	k=32	k=512	k=8k
twitter (d)	1 (1)	0.96 (1)	0.96 (1.17)
web-cc-fl (d)	1 (1)	0.96 (1)	0.96 (1.18)
kron_27 (d)	1 (1)	0.96 (1)	0.96 (1.05)

deteriorates in random ordering. CGR sees an average improvement of 1.13x with BP and its compression ratio drops to 0.61x with random ordering.

BFS Performance: Both EFG and CGR benefit from improved locality and are impacted negatively by random ordering. HALO improves EFG's performance by 1.32x and CGR's performance by 1.35x. Random ordering reduces the performance to 0.65x and 0.8x for the two methods. The runtime performance of BFS depends on factors such as memory coalescing, read amplification, prefetcher's effectiveness, etc., and locality friendly orderings improve these characteristics.

9.1.5 Sensitivity to Forward Quantum Size

We use a quantum value of $k = 512$ for forward pointers in this work. That is, there is a pointer for every 512 elements in a list. Forward pointers have minimal overhead, particularly for graphs with a power-law degree distribution. Since most nodes have a low degree, no pointers are stored for such lists. We show the variation in graph size and BFS's runtime for different values of k in Table. 9.2 where the values are normalized to the $k = 32$ case. Increasing the quantum beyond 512 has negligible (¡0.01) benefit in the total graph size. However, it adversely affects performance beyond 512.

9.1.6 Compression Time

Since compression is an offline preprocessing step for static graphs, it is not on the critical path, and optimising the compression performance was not a priority in this work. Nev-

ertheless, compressing graphs with EF is quite efficient and we were able to compress all the graphs in this work that were originally in the CSR format in less than 5 minutes on a 44 core Broadwell server. We use folly [102] for compressing individual neighbour lists, and OpenMP is used for coarse grained parallelism. It is possible to optimise this further by paralleling the compression within each list, and even the compression stage can be offloaded to a GPU. Compressing graphs with CGR using the reference implementation took more than 30 minutes for several graphs in the work.

9.2 Limitations and Discussion

The EF encoding scheme works well when the list of values being compressed are scattered randomly. However, in cases like web-graphs, we typically have long runs of contiguous values. This is, in fact, a good case for gap based encoding schemes. EF, on the other hand, does not benefit from smaller gaps. To use a motivating example from [112], consider a sequence $S = [0, 1, 2, ..., n-2, u-1]$ of length n where the first $n-1$ values are contiguous. This is a highly compressible sequence where the length of the run and the last value are sufficient to describe S. On the other hand, EF still uses $2 + \lceil \log u/n \rceil$ bits to encode each element, which is the same as any random sequence. The general approach to deal with this problem is to partition the sequence so that some partitions can be encoded more efficiently (e.g., with run-length coding). In PEF [112], the authors propose a method for selecting partition sizes in this model. A different optimisation for EF encoding uses clustering [113] to exploit similarities between lists. We did not incorporate these solutions in this work, but extensions to the EFG format are possible. One of the advantages of EFG is that maps well to GPUs due to its regular nature. There is usually a trade-off in compression ratio and decompression time, and it is an open question if we can improve both metrics simultaneously.

We found that the GPU used in this work lacks support for some types of bit-manipulation instructions that are present on current CPUs, and the memory alignment restrictions posed

by GPUs also introduce inefficiencies. For instance, although the lower-bits array can be accessed in constant time, it takes multiple instructions due to alignment restrictions. As a recommendation to hardware architects, improvements in these areas would be welcome. In the GPU setting with limited memory, graph compression is a complementary solution. There is always data (e.g., edge weights) that cannot be compressed easily. So compression does not preclude other hardware or software solutions that address the same problems.

9.3 Conclusion

Since real-world graphs are typically larger than a GPU's memory capacity, a variety of solutions can be used to address the challenges involved in graph analytics on GPUs. In this work, we looked at graph compression as a means to accommodate large graphs in GPU memory. Traditional graph compression schemes, while effective in compressing the graphs, cannot be used easily on GPUs due to the sequential and dependent nature of the decompression phase. We proposed the Elias Fano Graph (EFG) representation, based on Elias Fano encoding, as a GPU-friendly compressed graph representation that encompasses several desirable properties in terms of compression ratio and the decompression performance. The seemingly irregular problem of decompression can be broken down into common high performance primitives such as parallel scans and searches, which results in an efficient and load-balanced implementation for graph traversals. We showed that we can compress several large graphs by a factor of 1.5x and outperform out-of-core approaches in traversal runtime by a factor of 3.8x. Our implementation is also 2x faster than the current state of the art in GPU based compressed graph traversals. The techniques described here were designed for GPUs, but can also be extended to parallel CPU implementations. The code for this work is published at `https://github.com/pgera/efg`.

CHAPTER 10

CONCLUSION

As we reach the end of Moore's law, the focus of high performance architectures has shifted to domain specific accelerators. GPUs were conventionally used only for graphics and media applications, but the architecture and the programming model has evolved to accelerate scientific applications, machine learning, and increasingly a broad class of applications under the data analytics umbrella including graph analytics. On the one hand, GPUs have very high compute throughput and fast access to device memory, which enables them to exploit thread level parallelism and hide long latency operations. On the other hand, the capacity of device memory is limited and not sufficient to accommodate large datasets. We believe that his trend will also manifest in future accelerators, and we will have increasingly heterogeneous systems with variable amounts of compute and memory on devices.

In this dissertation, we looked at large graphs and graph analytics kernels for GPUs, which pose several challenges when the graphs do not fit in memory. Graph kernels are generally memory bound, have poor locality, and the memory access pattern is hard to predict. The conventional approach in such scenarios has been to scale the problem over multiple devices if a single device cannot accommodate it in memory. In this dissertation, we approached the problem from a different angle. We explored optimisations that can overcome the memory limitations of a single GPU. First, we analysed recent architectures which support a feature known as unified memory that lets us address host memory transparently from the GPU. This increases the amount of addressable memory, but the performance characteristics for graph kernels are quite poor in such a system. We proposed a light weight graph ordering method, HALO, to improve the memory access characteristics in the unified memory model. We also explored the connection between locality and compression, and showed that we can use techniques from graph compression to also improve

locality. Next, we looked at graph compression as an independent and a complementary approach. The main draw for graph compression is that if a graph can be compressed sufficiently to fit in memory, it avoids expensive accesses over the interconnect to the host or to other GPUs. We proposed a graph compression format, EFG, based on Elias-Fano encoding that performs well in terms of compression ratio and runtime decompression on GPUs. Graph compression has been studied extensively in the CPU context, but the GPU architecture makes is difficult to use most prior proposals directly. Our implementation exceeds the decompression performance of the current state of the art approach in GPU based compressed graph traversals and maintains a competitive compression ratio. Finally, we also looked at the relationship between graph compression and reordering, and showed that the two solutions can be combined. We can improve the runtime performance further by combining locality friendly orderings with graph compression. At the same time, the EFG representation is resilient to pathological orderings.

REFERENCES

[1] D. Ajwani, R. Dementiev, and U. Meyer, "A computational study of external-memory bfs algorithms," in *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, Society for Industrial and Applied Mathematics, 2006, pp. 601–610.

[2] A. Kyrola, G. E. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," USENIX, 2012.

[3] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, ACM, 2009, pp. 233–244.

[4] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "Mosaic: Processing a trillion-edge graph on a single machine," in *Proceedings of the Twelfth European Conference on Computer Systems*, ACM, 2017, pp. 527–543.

[5] M. Besta and T. Hoefler, "Survey and taxonomy of lossless graph compression and space-efficient graph representations," *arXiv preprint arXiv:1806.01799*, 2018.

[6] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[7] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proceedings of the 1969 24th national conference*, ACM, 1969, pp. 157–172.

[8] J. A. George, "Computer implementation of the finite element method," STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, Tech. Rep., 1971.

[9] H. Wei, J. X. Yu, C. Lu, and X. Lin, "Speedup graph processing by graph ordering," in *Proceedings of the 2016 International Conference on Management of Data*, ACM, 2016, pp. 1813–1828.

[10] M. Sha, Y. Li, and K.-L. Tan, "Gpu-based graph traversal on compressed graphs," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 775–792.

[11] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and

commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, Ieee, 2009, pp. 44–54.

[13] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray User's Group (CUG)*, vol. 19, pp. 45–74, 2010.

[14] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of mathematical sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[15] L. K. Fleischer, B. Hendrickson, and A. Pınar, "On identifying strongly connected components in parallel," in *International Parallel and Distributed Processing Symposium*, Springer, 2000, pp. 505–511.

[16] P. Harish and P. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *International conference on high-performance computing*, Springer, 2007, pp. 197–208.

[17] M. Hussein, A. Varshney, and L. Davis, "On implementing graph cuts on cuda."

[18] D. Merrill, M. Garland, and A. Grimshaw, "High-performance and scalable gpu graph traversal," *ACM Transactions on Parallel Computing*, vol. 1, no. 2, p. 14, 2015.

[19] L. Luo, M. Wong, and W.-m. Hwu, "An effective gpu implementation of breadth-first search," in *Proceedings of the 47th design automation conference*, ACM, 2010, pp. 52–55.

[20] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," *Scientific Programming*, vol. 21, no. 3-4, pp. 137–148, 2013.

[21] H. Liu and H. H. Huang, "Enterprise: Breadth-first graph traversal on gpus," in *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*, IEEE, 2015, pp. 1–12.

[22] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *ACM SIGPLAN Notices*, ACM, vol. 51, 2016, p. 11.

[23] O. Green and D. A. Bader, "Custinger: Supporting dynamic graph algorithms for gpus," in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, IEEE, 2016, pp. 1–6.

[24] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "Graphbig: Understanding graph computing in the context of industrial solutions," in *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*, IEEE, 2015, pp. 1–12.

[25] Y. Pan, Y. Wang, Y. Wu, C. Yang, and J. D. Owens, "Multi-gpu graph analytics," in *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, IEEE, 2017, pp. 479–490.

[26] E. Mastrostefano and M. Bernaschi, "Efficient breadth first search on multi-gpu systems," *Journal of Parallel and Distributed Computing*, vol. 73, no. 9, pp. 1292–1305, 2013.

[27] A. Gharaibeh, T. Reza, E. Santos-Neto, L. B. Costa, S. Sallinen, and M. Ripeanu, "Efficient large-scale graph processing on hybrid cpu and gpu systems," *arXiv preprint arXiv:1312.3018*, 2013.

[28] *Unified memory on pascal and volta*, `http://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf`, (Accessed on 02/20/2019).

[29] *Opencl20-api*, `https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf`, (Accessed on 02/20/2019).

[30] *Hmm-gup-no-more.pdf*, `https://xdc2018.x.org/slides/hmm-gup-no-more.pdf`, (Accessed on 02/20/2019).

[31] L. Dhulipala, G. E. Blelloch, and J. Shun, "Theoretically efficient parallel graph algorithms can be fast and scalable," *arXiv preprint arXiv:1805.05208*, 2018.

[32] F. Busato, O. Green, N. Bombieri, and D. A. Bader, "Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, IEEE, 2018, pp. 1–7.

[33] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2011, p. 65.

[34] NVIDIA Corp., *NVIDIA Tesla P100*, `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`, 2016.

[35] ——, *NVIDIA Tesla V100*, http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2016.

[36] ——, *NVIDIA Driver Downloads*, https://www.nvidia.com, 2018.

[37] Peng Wang, *UNIFIED MEMORY ON P100*, https://www.olcf.ornl.gov/wp-content/uploads/2018/02/SummitDev_Unified-Memory.pdf, 2017.

[38] N. Akhenykh, *Unified Memory On Pascal and Volta*, http://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf, 2017.

[39] ——, *Everything You Need to Know About Unified Memory*, http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf, 2018.

[40] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, "Towards High Performance Paged Memory for GPUs," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[41] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, "Mosaic: A GPU Memory Manager with Application-transparent Support for Multiple Page Sizes," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2017.

[42] P. Boldi and S. Vigna, "The WebGraph Framework I: Compression Techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, 2004.

[43] P. Boldi, A. Marino, M. Santini, and S. Vigna, "BUbiNG: Massive Crawling for the Masses," in *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web*, 2014.

[44] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "UbiCrawler: A Scalable Fully Distributed Web Crawler," *Software: Practice & Experience*, vol. 34, no. 8, pp. 711–726, 2004.

[45] J. Leskovec and A. Krevl, *SNAP Datasets: Stanford Large Network Dataset Collection*, http://snap.stanford.edu/data, Jun. 2014.

[46] J. Sybrandt, M. Shtutman, and I. Safro, "MOLIERE: Automatic Biomedical Hypothesis Generation System," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '17, 2017.

[47] R. A. Rossi and N. K. Ahmed, "The Network Data Repository with Interactive Graph Analytics and Visualization," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[48] M. R. Garey and D. S. Johnson, *Computers and intractability*, vol. 29.

[49] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan, "On compressing social networks," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 2009, pp. 219–228.

[50] L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, and A. Shalita, "Compressing graphs and indexes with recursive graph bisection," *arXiv preprint arXiv:1602.08820*, 2016.

[51] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, "Rabbit order: Just-in-time parallel reordering for fast graph analysis," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*, IEEE, 2016, pp. 22–31.

[52] Y. Lim, U Kang, and C. Faloutsos, "Slashburn: Graph compression and mining beyond caveman communities," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 12, pp. 3077–3089, 2014.

[53] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th international conference on World wide web*, ACM, 2011, pp. 587–596.

[54] A. Apostolico and G. Drovandi, "Graph compression by bfs," *Algorithms*, vol. 2, no. 3, pp. 1031–1044, 2009.

[55] E. Lee, J. Kim, K. Lim, S. H. Noh, and J. Seo, "Pre-select static caching and neighborhood ordering for bfs-like algorithms on disk-based graph engines," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA: USENIX Association, Jul. 2019, pp. 459–474, ISBN: 978-1-939133-03-8.

[56] S. Han, L. Zou, and J. X. Yu, "Speeding up set intersections in graph algorithms using simd instructions," in *Proceedings of the 2018 International Conference on Management of Data*, ACM, 2018, pp. 1587–1602.

[57] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, "Making caches work for graph analytics," in *2017 IEEE International Conference on Big Data (Big Data)*, IEEE, 2017, pp. 293–302.

[58] V. Balaji and B. Lucia, "When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, Los Alamitos, CA, USA: IEEE Computer Society, 2018, pp. 203–214.

[59] P. Faldu, J. Diamond, and B. Grot, "A closer look at lightweight graph reordering," *2019 IEEE International Symposium on Workload Characterization (IISWC)*, 2019.

[60] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré, "Emptyheaded: A relational engine for graph processing," *ACM Transactions on Database Systems (TODS)*, vol. 42, no. 4, p. 20, 2017.

[61] G. Karypis and V. Kumar, "A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," 1998.

[62] A. Bavelas, "Communication patterns in task-oriented groups," *The Journal of the Acoustical Society of America*, vol. 22, no. 6, pp. 725–730, 1950.

[63] G. Sabidussi, "The centrality index of a graph," *Psychometrika*, vol. 31, no. 4, pp. 581–603, 1966.

[64] D. Eppstein and J. Wang, "Fast approximation of centrality," in *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, Society for Industrial and Applied Mathematics, 2001, pp. 228–229.

[65] S. Milgram, "The small world problem," *Psychology today*, vol. 2, no. 1, pp. 60–67, 1967.

[66] Y. Rochat, "Closeness centrality extended to unconnected graphs: The harmonic centrality index," Tech. Rep., 2009.

[67] M. Garg, "Axiomatic foundations of centrality in networks," 2009.

[68] *Closeness centrality in networks with disconnected components — tore opsahl*, `https://toreopsahl.com/2010/03/20/closeness-centrality-in-networks-with-disconnected-components/`, (Accessed on 03/29/2019).

[69] P. Boldi and S. Vigna, "Axioms for centrality," *Internet Mathematics*, vol. 10, no. 3-4, pp. 222–262, 2014.

[70] E. Angriman, "Efficient computation of harmonic centrality on large networks: Theory and practice," Master's thesis, University of Padova, 2016.

[71] P. Boldi and S. Vigna, "In-core computation of geometric centralities with hyperball: A hundred billion nodes and beyond," in *2013 IEEE 13th International Conference on Data Mining Workshops*, IEEE, 2013, pp. 621–628.

[72] J. Mackenzie, A. Mallia, M. Petri, J. S. Culpepper, and T. Suel, "Compressing inverted indexes with recursive graph bisection: A reproducibility study," in *Proc. ECIR*, 2019, pp. 339–352.

[73] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell system technical journal*, vol. 49, no. 2, pp. 291–307, 1970.

[74] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail, "Approximating betweenness centrality," in *International Workshop on Algorithms and Models for the Web-Graph*, Springer, 2007, pp. 124–137.

[75] J. D. Ullman and M. Yannakakis, "High-probability parallel transitive-closure algorithms," *SIAM Journal on Computing*, vol. 20, no. 1, pp. 100–125, 1991.

[76] W.-M. Chan and A. George, "A linear time implementation of the reverse cuthill-mckee algorithm," *BIT Numerical Mathematics*, 1980.

[77] A. George and J. W. Liu, "An implementation of a pseudoperipheral node finder," *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 284–295, 1979.

[78] J.-C. Luo, "Algorithms for reducing the bandwidth and profile of a sparse matrix," *Computers & structures*, vol. 44, no. 3, pp. 535–548, 1992.

[79] G. K. Kumfert, "Object-oriented algorithmic laboratory for ordering sparse matrices," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2000.

[80] K. I. Karantasis, A. Lenharth, D. Nguyen, M. J. Garzarán, and K. Pingali, "Parallelization of reordering algorithms for bandwidth and wavefront reduction," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE Press, 2014, pp. 921–932.

[81] A. Mallia, M. Siedlaczek, J. Mackenzie, and T. Suel, "PISA: performant indexes and search for academia," in *Proceedings of the Open-Source IR Replicability Challenge co-located with 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval, OSIRRC@SIGIR 2019, Paris, France, July 25, 2019.*, 2019, pp. 50–56.

[82] S. Beamer, K. Asanovic, and D. Patterson, *The GAP Benchmark Suite*, 2015. arXiv: 1508.03619 [cs.DC].

[83] U. Meyer and P. Sanders, "Delta-stepping: A parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, no. 1, pp. 114–152, 2003.

[84] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel gpu methods for single-source shortest paths," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IEEE, 2014, pp. 349–359.

[85] C. Li, R. Ausavarungnirun, C. J. Rossbach, Y. Zhang, O. Mutlu, Y. Guo, and J. Yang, "A framework for memory oversubscription management in graphics processing units," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2019, pp. 49–63.

[86] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim, "Batch-aware unified memory management in gpusfor irregular workloads," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2020.

[87] S. W. Min, V. S. Mailthody, Z. Qureshi, J. Xiong, E. Ebrahimi, and W.-m. Hwu, "Emogi: Efficient memory-access for out-of-memory graph-traversal in gpus," *Proceedings of the VLDB Endowment*, vol. 14, no. 2, pp. 114–127, 2021.

[88] P. Gera, H. Kim, P. Sao, H. Kim, and D. Bader, "Traversing large graphs on gpus with unified memory," *Proceedings of the VLDB Endowment*, vol. 13, no. 7, pp. 1119–1133, 2020.

[89] A. H. N. Sabet, Z. Zhao, and R. Gupta, "Subway: Minimizing data transfer during out-of-gpu-memory graph processing," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[90] *Cugraph - rapids graph analytics library*, https://github.com/rapidsai/cugraph, (Accessed on 10/02/2020).

[91] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE micro*, vol. 28, no. 2, pp. 39–55, 2008.

[92] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015.

[93] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer, "Graph structure in the web—revisited: A trick of the heavy tail," in *WWW Companion*, 2014, pp. 427–432.

[94] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[95] D. Culler, J. P. Singh, and A. Gupta, *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing, 1999.

[96] P. Elias, "Efficient storage and retrieval by content and address of static files," *Journal of the ACM (JACM)*, vol. 21, no. 2, pp. 246–260, 1974.

[97] R. M. Fano, "Efficient storage and retrieval by content and address of static files," *Computation Structures Group Memo. Massachusetts Institute of Technology, Project MAC*,

[98] S. Vigna, "Quasi-succinct indices," in *Proceedings of the sixth ACM international conference on Web search and data mining*, ACM, 2013, pp. 83–92.

[99] A. Mallia, *Sorted integers compression with elias-fano encoding*, `https://www.antoniomallia.it/sorted-integers-compression-with-elias-fano-encoding.html`, (Accessed on 07/27/2020).

[100] D. R. Clark and J. I. Munro, "Efficient suffix trees on secondary storage," in *SODA*, vol. 96, 1996, pp. 383–391.

[101] S. Vigna, "Broadword implementation of rank/select queries," in *International Workshop on Experimental and Efficient Algorithms*, Springer, 2008, pp. 154–168.

[102] *Folly: An open-source c++ library developed and used at facebook.* `https://github.com/facebook/folly`, (Accessed on 08/04/2020).

[103] M. Bisson, M. Bernaschi, and E. Mastrostefano, "Parallel distributed breadth first search on the kepler architecture," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 7, pp. 2091–2102, 2015.

[104] O. Green, R. McColl, and D. A. Bader, "Gpu merge path: A gpu merging algorithm," in *Proceedings of the 26th ACM international conference on Supercomputing*, 2012, pp. 331–340.

[105] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for cuda," in *GPU computing gems Jade edition*, Elsevier, 2012, pp. 359–371.

[106] D. Merrill, "Cub," *NVIDIA Research*, 2015.

[107] G. Buehrer and K. Chellapilla, "A scalable pattern mining approach to web graph compression with communities," in *Proceedings of the 2008 International Conference on Web Search and Data Mining*, 2008, pp. 95–106.

[108] P. Boldi and S. Vigna, "The webgraph framework i: Compression techniques," in *Proceedings of the 13th international conference on World Wide Web*, ACM, 2004, pp. 595–602.

[109] J. Shun, L. Dhulipala, and G. E. Blelloch, "Smaller and faster: Parallel processing of compressed graphs with ligra+," in *2015 Data Compression Conference*, IEEE, 2015, pp. 403–412.

[110] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 135–146.

[111] M. Besta, D. Stanojevic, T. Zivic, J. Singh, M. Hoerold, and T. Hoefler, "Log (graph): A near-optimal high-performance graph representation.," 2018.

[112] G. Ottaviano and R. Venturini, "Partitioned elias-fano indexes," in *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, ACM, 2014, pp. 273–282.

[113] G. E. Pibiri and R. Venturini, "Clustered elias-fano indexes," *ACM Transactions on Information Systems (TOIS)*, vol. 36, no. 1, p. 2, 2017.